CONNECTSIM++: A GENERAL PURPOSE CONNECTIONIST NETWORK SIMULATOR

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

By
B. G. MOHANA KRISHNA

to the
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
JANUARY, 1991

CERTIFICATE

It is certified that the work contained in the thesis entitled "CONNECTSIM++ : A General Purpose Connectionist Network Simulator" by *B. G. Mohana Krishna*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(H. Karnick)
Assistant Professor
Department of CSE

January, 1991
I.I.T Kanpur

ABSTRACT

ConnectSim++ is the result of an effort to provide an efficient general-purpose connectionist simulation tool. It has incorporated in its implementation the advantages that a simulation tool of its nature can gain from adopting an object-oriented approach, while not compromising on the efficiency and speed of the system. The design philosophy of the system exploits the power that derives from allowing the user complete freedom to state the idiosyncrasies of the problem being solved, while providing him with an environment that has all the tools to handle the more general aspects of it, such as the definition of a network and its simulation. Several simulations are carried out to substantiate the previous statement.

ACKNOWLEDGEMENTS

Working on this project has been a very stimulating and rewarding experience. The credit for this is due mainly to Dr. Karnick, my thesis supervisor. I take this opportunity to express my heartfelt thanks to him - for suggesting the project, for the ideas and guidance, for the encouragement and the freedom that he provided. The thesis, with respect to both its form and content, owes a lot to his interest.

Bringing out this report in time and in its present form would not have been possible but for help from several friends who considered the work as their own. When I seemed almost lost due to my inexperience with any document preparation package, C.A. Shankara took it upon himself to get the report in print and saw it to completion despite his own pressing work. K.G.Shastri has patiently spent several hours on the figures. He never fails to find time to do a favor to someone. Mahesh has relived me of a major share of the burden by doing the typing and then proof-reading the chapters. Ananda, as is usual with him, has been of great help in proof-reading and in getting the final copies ready for submission. I would have been terribly handicapped but for cheerful help by each one of them. Rajashekhar, Suresh, Prabhu, Ram Mohan, Srinivas and many other friends have been helpful in various stages of the project work and during my stay here. My sincere thanks to all these friends who have made me wish I had stayed longer.

C O N T E N T S

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Connectionist models have evoked great interest as a new computing paradigm which offers the hope of attacking long standing problems in AI related to human cognitive abilities. These have been variously labelled as *Artificial Neural Networks*, *Parallel Distributed Processing* models and *Neuromorphic* systems. These models are characterized by a very large number of simple computational elements operating in parallel and interconnected in complex ways reminiscent of biological neural nets. Hence they are believed to inherit several attributes of the animal brain, mainly the ability to explore a large number of competing hypotheses in parallel. These models have been studied over the past several years, and the volume of research in the field over this period has been substantial. The results have been encouraging; if they have not been as spectacular as was made out, the reasons may be attributed to want of right technology and an insufficient understanding of the mechanisms involved, rather than to the weakness of the paradigm itself. But the promise of potential continues to be as exciting as ever.

1.1 The Neural Inspiration for Connectionist Models:

Despite the tremendous success of the *stored program* approach of the von Neumann architecture in the performance of well-structured and unambiguously specified tasks, the efficacy of the model with regard to tasks which are carried out on a day-to-day basis by humans raises serious doubts. Such tasks include, among many others, the ability to recognize a human face and, on so doing, recollecting a host of related information such as the way he speaks, his profession etc., the ability to retrieve and manipulate data such as images, sounds, smells, sensations and thoughts, and the ability to bring into play a huge bank of information, retrieving from it, using some while discarding others in quick succession to arrive at a decision - all that we

seem to do so naturally and effortlessly. Some of these abilities are also shared by the supposedly less intelligent animals as well. To be capable of general cognitive processes such as search, classification, learning and hypothesis discovery which endow us with the above-mentioned abilities, the human brain (more generally, the animal brain) must be employing an architecture that is radically different from that of the conventional computer to represent, process, and retrieve data from a large database built from experience through interaction with the environment. This view seems vindicated by the characteristics of the hardware of the brain as revealed by the studies in the related disciplines.

To perform a simple task such as recognizing a picture, the human brain takes less than a second. Considering the fact that the neuron, which is the basic computational building block in the brain, has firing rates of the order of a millisecond, there might be about a hundred computational time steps involved in the completion of the task. The best AI program for a similar task, running on a computer with components having nanosecond switching speeds, requires of the order of a million computational steps. This hundred step rule of the brain is a typical constraint on any model of behavior, and brings forth several possibilities about the characteristics and the functioning of the brain which might offer valuable clues in the quest for an alternative paradigm for computation:

(i) The brain is made up of an extremely large number of ($\approx 10^{11}$) neurons which are essentially very simple in their construction and with switching speeds of the order of a millisecond.

(ii) The superiority of the animal brain arises from the complex interconnectedness of the neurons. The fan-in and fan-out of a neural unit is typically of the order of 10^4 i.e. each neuron receives inputs from about 10,000 other neurons and feeds about 10,000 others. (For conventional chips, a

connectivity of about 6-8 is considered quite difficult to achieve.)

(iii) The timing considerations also go to show that the amount of information exchange between neurons is very small, a few bits at most. This means that direct exchange of complex structures is precluded and, if present, must be encoded in some way. This leads to the possibility that the critical information is captured in the connections between units.

(iv) The process of learning modifies the connection strengths between the individual units.

The above observations, in turn, lead to some very important implications which can be crucial to the design of artificial neural net models:

(i) These networks for computation are not *programmed* in the conventional sense, rather they *learn* to solve problems through interaction with the environment.

(ii) Very little computation is carried out at the site of an individual node. There are no explicit memory or processing locations in neural networks, but are implicit in the connections between the nodes.

(iii) There is no *overall supervisor* or a *central executive*, each node having the same simple features. Control is thus distributed and not central. The combined effect of many such nodes acting in concert gives neural networks their power.

(iv) Not all sources of input feeding a node are of equal importance. So, a weight - a measure of the importance - is associated with each connection. Depending on the kind of influence each input has on the node, excitatory or inhibitory, weights can be positive or negative. It is by varying the connection strengths and the sign that the network learns.

(v) The inputs arriving at a node are transformed according to the node's *transfer function (activation function)*, often a simple one such as the sigmoidal function. The function is generally of a type which is nonlinear and continuous. (With a linear function the advantages of having multiple layers are lost. Also, if the function is discontinuous, and hence non-differentiable, it interferes with the network's ability to learn and generalize.)

(vi) Since all of the connections can carry signals simultaneously and all of the processing elements can act in parallel to integrate their arriving data, such a network can bring a large amount of knowledge to bear simultaneously when making a decision, and can weigh many choices at once. The massive parallelism provided by the large number of neurons and the connections between them is used in some systems to implement a sort of simultaneous brute-force search through individual items in a large knowledge base and in others to allow richer representations; the pattern of activity over a large group of units represents an item, and different items are represented by alternative patterns of activity over the same set of units. Given an initial state of the network and some inputs, one of the patterns will emerge finally as the answer.

The above property whereby a single entity is represented by a pattern of activity spread over many computing elements and each computing element is involved in representing many different entities is termed *Distributed Representation*. This is in contrast to the alternative scheme of *Local Representation* wherein the concepts and the representing units stand in a one-to-one relation. The latter is a characteristic of the conventional digital computer.

The distributed representation has several merits which make it very desirable:

(a) *Associative Memory*: Provides a highly efficient associative memory which enables recall of items in their entirety from a partial description of their contents. Such a scheme is very difficult to implement on a conventional machine because access to items requires a precise knowledge of their location, and the recall of an item that best fits the pattern specified by the partial description requires a massive search of the whole entity space.

(b) *Graceful Degradation*: What is more interesting about the distributed representation is that the recall often turns out correct even if a few of the partial descriptions are wrong. By extension, the network also functions correctly even if a few units involved in the representation malfunction. In the worst case, the resulting pattern is imperfect but still usable. Each macroscopically important behavior of the network is implemented by a large number of distinct microscopic units, so that loss of any small random subset will result in little or no change to a macroscopic description of the network behavior. Such graceful degradation with damage or overloading implies that no special error recovery mechanism is required and is a natural byproduct of the nature of the retrieval mechanism.

(c) *Similarity and Generalization*: The networks are trained on data for which the right answer is known, after which they should be able to generalize what they know, responding correctly to new data. This is a very important requirement when such networks are used as *pattern classifiers*. The network should be able to respond with correct outputs even for inputs that deviate slightly from the pattern that it had been taught previously.

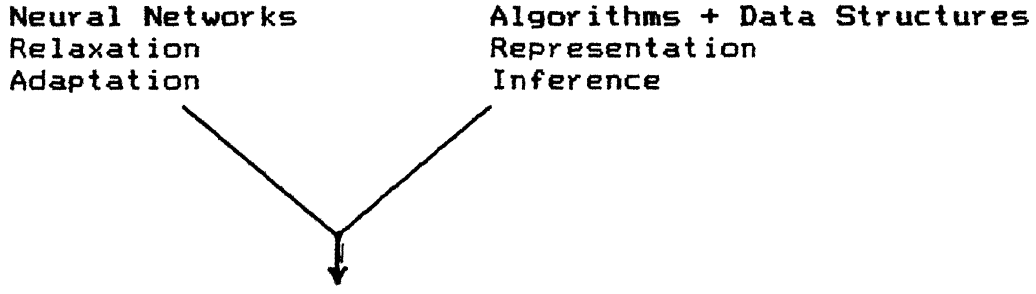
One of the major problems of distributed representations is "cross-talk" among the stored concepts. To add a simple new piece of macroscopic knowledge to the network it is necessary to change

the interactions between many microscopic units slightly so that their joint effects implement the new knowledge without loss of the knowledge that is already stored. If there is an interaction between two stored patterns that represent two distinct concepts, it is possible that modifications caused by the addition of a new piece of knowledge might result in the wrong pattern being output.

Orthogonal patterns may be used to represent each of the concepts that the network is required to store, so that interference between patterns is altogether eliminated. This is not always desirable, however, since it eliminates one of the very interesting capabilities of the network - that of automatic generalization.

1.2 The Promise of the Connectionist Paradigm:

Connectionist models can be viewed as synthesizing two traditionally opposed approaches to AI, as suggested by the figure below:



One focuses on the parallelism, robustness and plasticity of animal brains and aims at exploring ways of automatically generating high performance. In this mode *relaxation* is the dominant mode of computing in which the system *settles into a solution* rather than *calculating a solution.* The system continually attempts to adapt itself to respond to inputs from the environment.

The other approach concentrates on the detailed structure of tasks, data structures to represent them in the conventional computer notation and algorithms to generate the solution or to draw inferences from the facts stored in the knowledge base. Structured connectionist models attempt to capture the best of both paradigms. One of the central goals of research into connectionist models has been to design intelligent machines capable of autonomous learning and skilful performance of tasks in the presence of noisy and distorted data within complex environments that are not under strict external control. Connectionist models with their attractive properties of self-organization, robustness, ability to generalize etc. have demonstrated their suitability for solving such problems. Several variants based on the connectionist approach have been proposed and built, and have proved their potential at tasks such as signal processing (as adaptive filters), adaptive learning from examples, pattern recognition and classification, combinatorial optimization, language understanding and inference with varying degrees of success.

Though it is inconceivable that neural net based architectures will completely replace conventional computers, there is no denying the prospect of these architectures greatly augmenting their capabilities. A remarkable fact about computer architecture has been that special purpose architectures, in general, have had very little impact and that should be a caution against excessive optimism in favor of connectionist models. However, presently efforts are on to combine the power of neural nets with those of digital computers by integrating circuits that implement a neural network into a personal computer and by providing software environments that enable their effective exploitation.

A large amount of research effort is also directed at developing new models that implement more powerful and more general learning algorithms and other theoretical aspects of connectionist models. Some of the important issues being

investigated into, include:

-the stability-plasticity aspects and noise sensitivity of the learning process.
-combining supervised and unsupervised learning.
-time and sequence problems in distributed decision making.
-whether specific models are more appropriate for a given class of computation than other models.
-the hardware that best suits particular neural net models.
-the representation of complex concepts.

Much of the present effort is dedicated to building models of intelligent activity. Advantages of this approach include its link to natural intelligence, increased noise immunity and potential ease of implementation on parallel hardware. By taking seriously the computational constraints faced by nature, and the structure of tasks known from artificial intelligence and other disciplines, attempts are being made to discover algorithms employed by animals that might be effective for machines. The studies being carried out might also provide new insights on how to do parallel computing and the fundamental properties of highly parallel computation.

1.3 Simulation of Connectionist Models:

Modeling has been an important aspect in the study of any complex real-life system, and all the more so with artificial neural networks. A direct study of the low level processes involved in functioning of the animal brain has not been feasible mainly due to the complexity of its composition and structure, and a general lack of understanding of the underlying mechanisms that lead to the high level behavior. Since the aim is to understand the principles of the massively parallel computation, attempts have been made to abstract and simplify the problem sufficiently to get practically meaningful results.

Computer simulations of neural networks has proved to be a powerful tool in modeling. Researchers experimenting with structured connectionist networks have relied on computers to implement and test their ideas. Computer simulation has made possible the investigation of questions not resolvable by physiological, behavioral or formal approaches alone. It has also enabled the observation of how interactions between large, unstructured collections of simple computing units generate emergent properties that lead to what we term intelligent behavior. Mathematical models involving large and hierarchically organized systems of nonlinear ordinary differential equations of neurally based processes have been proposed. However the output of such complex systems are difficult, or in some cases impossible, to determine through means other than "running" the model in a numerical computer simulation. Even in cases where such complex mathematical models are not involved, the observation of the dynamics of the system as learning proceeds or as the system stabilizes has provided valuable insights and this was rendered possible by computer simulations. Such simulations have allowed much flexibility in respect of observing changes in the system behavior with changes in various parameters, the interaction between the parameters and tuning the parameters to achieve optimum system performance.

A researcher using computer simulation spends a good deal of his time in writing programs for this purpose. Also, it is a matter of common observation that the basic algorithms of the connectionist network are relatively easy to program, but that these rather simple "core" programs are of little value unless they are embedded in a system that lets the researcher observe and interact with their functions. These user interface programs are generally very tedious and time consuming to write. Separate programs written for individual systems involve duplication of programming effort due to many common subtasks that are performed in the course of specification, simulation and subsequent testing of the individual models. Also programs directed toward one particular system can be quite inflexible making it difficult to

modify the network being studied. Hence a general purpose simulator that abstracts and incorporates common functions performed in the construction, simulation and observation of a variety of networks, and further provides facilities whereby a user can easily specify features that are peculiar to the network being simulated, would simplify the overall process to a large extent. Such a simulation tool must also provide a powerful user-interface that insulates him from unnecessary detail extraneous to the problem.

1.4 The Present Simulator:

The motivation for the development of *ConnectSim++* came from SNAIL - another connectionist network simulator already existing at IIT, Kanpur. It too is a general-purpose simulator with several desirable properties such as its object-oriented paradigm, its flexibility to cater to a wide variety of models and ease of extensibility. Its main drawback is that it is very slow, having been implemented in LISPTALK [Mani and Srinivasaragahavan,1988]- a Lisp implementation of a SmallTalk-like language. The multiple levels of interpretation involved in the execution of programs made it almost impractical for use with networks of size beyond a certain limit. Even for reasonably small sizes, the simulation would take several hours. Also, the user required some time and effort to acquire familiarity with the unconventional syntax of the language used for network description and user-specified functions owing to its roots in LISPTALK.

More specifically, the aim was to design a connectionist simulation tool, sufficiently general-purpose not only to encompass currently known models but also capable of being extended to cover any new models that might be proposed. The extensibility of any system tool is aided if it is possible for the user to integrate his programs and data structures into it with ease. It was felt that the Rochester Connectionist Simulator [Goddard et al.,1989], implemented at the University of Rochester and being used for most of the research in connectionist networks there, by virtue of having been designed with similar objectives,

was a good model. Having been implemented in C it is known to be quite fast. At the same time, it was also difficult to give up the advantages offered by the object orientation of SNAIL.

Hence, it was decided to integrate all these objectives into ConnectSim++ by using C++, with its object-oriented features as the vehicle for implementation. ConnectSim++ relies on the concepts of *Encapsulation* and *Inheritance* provided by C++. Encapsulation is the modeling of system elements so that only the specification of the structure and behavior is visible to the user. The details of implementation which are of no concern to the user of the system are effectively inaccessible. Encapsulation, apart from providing a methodology for decomposing complex problems into simple, independent pieces, also enables the development of the system in distinct stages through effective modularization.

The property of inheritance allows the system user to represent hierarchical relationships among objects by defining a tree of object classes. Each child class inherits the attributes and behavior of its parent class and can easily augment them to define more specialized versions of the object represented by the parent class. This imparts to the system a high degree of flexibility and extensibility. Any enhancement or modification to the system is easily achieved through derived classes of the objects in question, which contain members that implement the enhancement together with the members of the parent class.

All system objects such as units, sites and links have been implemented as classes. Each object belongs to a particular class and has its own private memory to hold values associated with it. Access to these private members is restricted and is only through member functions declared in the public part of the class. The member functions implement methods to store, retrieve and manipulate the data contained in the private section of the class.

The use of the object-oriented paradigm in ConnectSim++ has resulted in a neat and highly modular system as was the case with its predecessor, SNAIL.

An overview of what is to follow in the rest of the book is relevant here. Chapter 2 introduces the user to the theory of connectionist networks, their connection topologies and several learning algorithms that are commonly used. Chapter 3 describes at length the organization of ConnectSim++, its features and a few important issues about its implementation. Chapter 4 illustrates the versatility and flexibility of the system with a set of simulations that cover a wide range of connectionist models. The final chapter discusses possible enhancements and improvements to the system that will increase its power and user-friendliness. Appendix A provides a comprehensive Users' Manual for the system and Appendix B contains the code for the simulations described in Chapter 4.

CHAPTER 2
CONNECTIONIST NETWORKS

2.1 Evolution:

The evolution of connectionist networks can be demarcated into two distinct phases. The major contributions in the first phase of the history of neural net research were by McCulloch and Pitts[1943], Hebb[1949], Rosenblatt[1962], Widrow[1960] and others. Minsky and Papert's[1969] analysis of the limitations of the capabilities of Rosenblatt's Perceptron model dampened all the initial enthusiasm and excitement for artificial neural nets, and brought to a standstill all work in the area for several years. However, more recent work by Hopfield[1982], Rumelhart and McClelland[1986], Hinton[1984,1987], Grossberg[1978],Feldman[1982] and others has led to a new resurgence of the field. This can be attributed to the development of new network topologies and learning algorithms, availability of powerful tools for testing them and new analog VLSI implementation techniques for connectionist models. Special purpose computing elements such as the Cognitron and Neocognitron have generated tremendous interest by their success with practical problems in pattern recognition.

2.2 Classification of Connectionist Networks:

Notwithstanding the fact that all the connectionist models share some common characteristics which make them stand apart from the other models of computation, a wide variety of them with different flavors and diverse capabilities have been proposed in the literature. Several criteria exist based on which neural nets may be classified. These include the characteristics of the individual units that constitute the network, the network topology, the type of inputs that are handled and the training algorithms employed.

One broad classification divides the network models into those trained under supervision and those trained without

supervision. Networks trained with supervision, such as the Hopfield net and the multilayer perceptron model, use sets of vector pairs during training. One, of each pair in the set, represents the inputs to the network and the other vector represents the corresponding desired output. Thus it is required that the correct output be known for all possible input patterns. The input vectors are presented to the network in sequence, the corresponding output vectors are clamped on the target unit set and a training rule such as the back-propagation algorithm is applied to adjust inter-node connection strengths and effect learning.

Networks trained without supervision are presented with patterns representing the input data, and self-organize without any external guidance. There is no *correct answer* for an input or a set of inputs. The weights are readjusted as some function of the current weight and the input signal. The network converges on a *feature-map* of the data used to train it. Examples of such networks include competitive learning networks and networks forming Kohonen feature-maps[Kohonen,1984], and are used as vector quantizers or to form clusters.

For a more detailed discussion of connectionist network taxonomy and examples of networks of different categories see [Lippmann,1987].

2.3 A General Framework for Connectionist Models:

The rest of this chapter tries to capture the essence of all these models and present a unified view, so that the various models can be considered as special cases of a general framework [Rumelhart, Hinton & McClelland, 1986]. The network model used by ConnectSim++ closely corresponds to this framework. The exact model used by the simulator is described in the next chapter, in the light of the following discussion.

We begin by dividing any connectionist or PDP model into

eight major components:

* A *set of processing units.*
* A *pattern of connectivity* among units.
* A *state of activation.*
* An *output function* for each unit.
* A *propagation rule* for propagating patterns of activities through the network of connectivities.
* An *activation rule* for combining the inputs impinging on a unit with the current state of that unit to produce a new level of activation for the unit.
* A *learning rule* whereby patterns of connectivity are modified by experience.
* An *environment* within which the system must operate.

Below is a description of these items, occasionally interspersed with other issues of importance relating to connectionist models.

2.3.1 Overview:

Figure 2.1 illustrates the basic aspects of connectionist systems. The units are represented as circles with connections between them. At any point in time the activation value of unit i is denoted $a_i(t)$. The connections pass the output value to other units in the system. The weight of a connection from unit i to unit j is represented as w_{ji}. At each input, the corresponding weight and the output value from the unit at the other end of the connection are combined (generally as a product of the two), and the net input for each type of input is then obtained as some simple function (very often the sum) of the individual inputs. The output of the unit is, in turn, obtained as a simple function of the activation value.

Any network may be assumed to be made up of units of one or more types. Units of the same type have the same type and number of inputs and outputs, and use the same rules for computing the potential and activation values.

2.3.2 Processing Units:

Any connectionist model begins with a set of processing units. Specifying the set of processing units and what they represent is typically the first stage of specifying a PDP model.

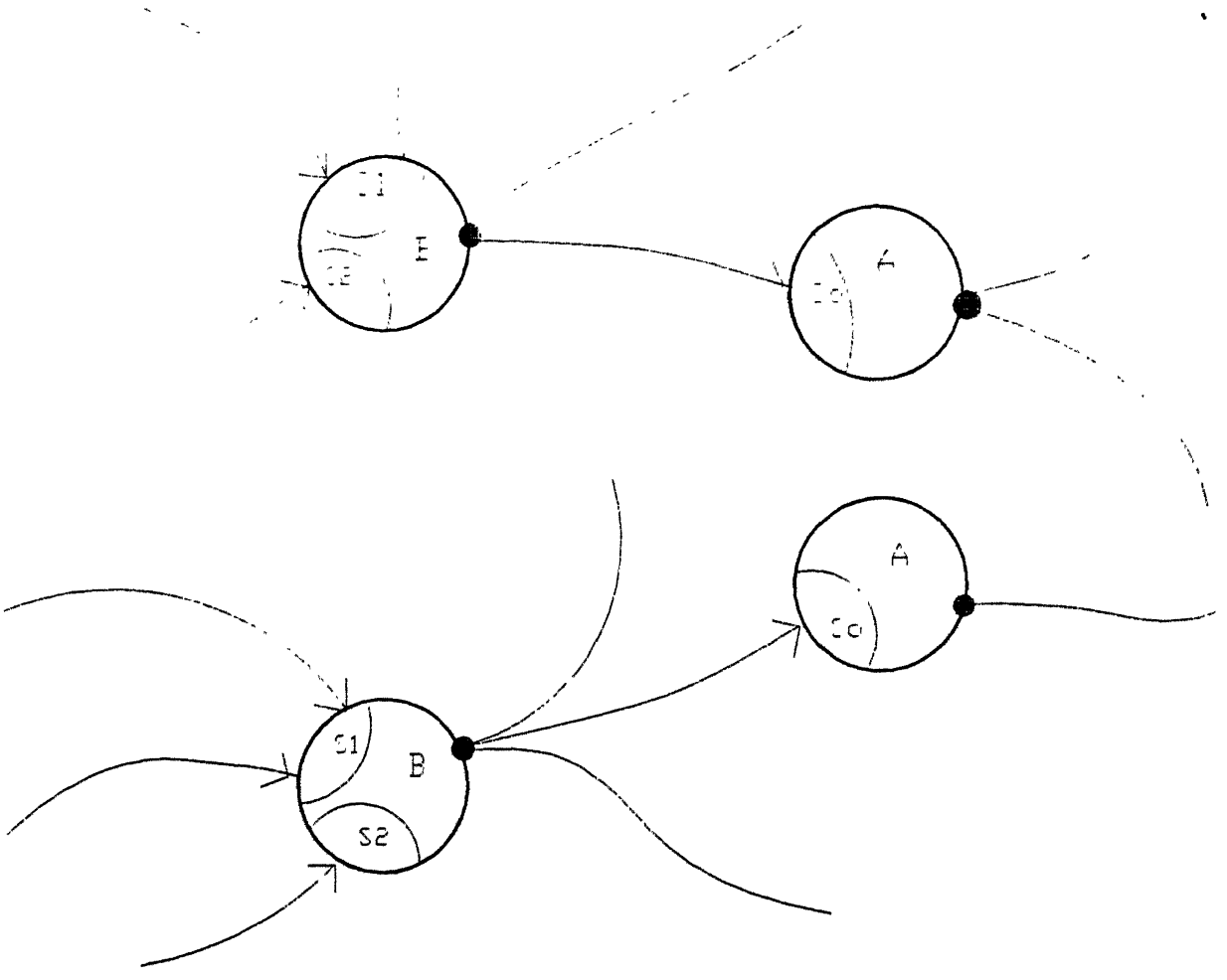


Figure 2.1 A Connectionist Network

In localized representations, each unit may represent a particular conceptual object (such as letters, words, features, concepts, etc.). When we speak of distributed representations, units are simply abstract elements over which meaningful patterns can be defined. A conceptual entity is then represented by the pattern of activation over a whole group of units - individual units may mean nothing.

For the purpose of analysis, it is usual to impose an arbitrary ordering on the units and designate the i^{th} unit u_i. It is also possible to group units of the same type (the type of a unit is decided by the number and kind of inputs and outputs, the function used to compute the output(s) from a given set of inputs, etc.) into one-, two-, or three-dimensional arrays. The latter method has the advantage of grouping units into blocks or layers which may be helpful in visualizing the network.

A unit's job is simply to receive input from its neighbors and, as a function of the inputs it receives, to compute an output value which it sends to its neighbors. The system is inherently parallel in that many units can carry out their computations at the same time.

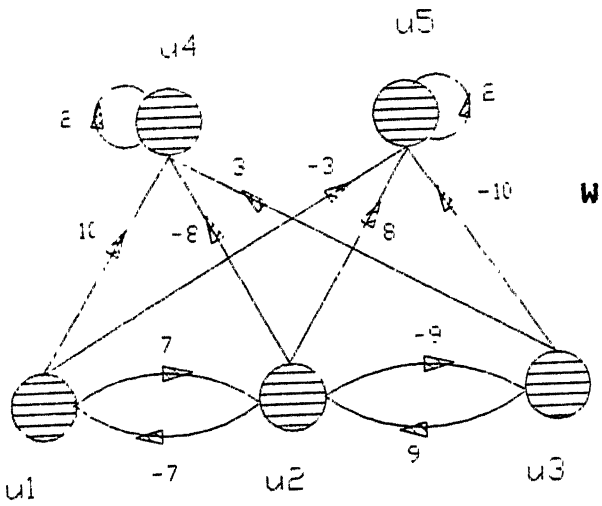
In any connectionist system, three types of units are usually recognized: input, output, and hidden. *Input units* receive inputs from sources external to the system under study. The *output units* send signals out of the system. The *hidden units* are those whose only inputs and outputs are within the system we are modeling. They are not *visible* outside the system.

2.3.3 Connectivity Pattern:

Units are connected to one another. Connection topology constitutes part of what the system knows and determines how it will respond to any arbitrary input. As mentioned in the previous chapter, specifying the pattern of connectivity among the processing units is equivalent to specifying the processing system and the knowledge encoded therein.

A pattern of connectivity can be represented by a *weight matrix* **W**, in which w_{ij} represents the strength and sense of the connection from unit u_j to unit u_i. The absolute value of w_{ij} signifies the *strength of the connection*; the sign of w_{ij} determines the excitatory(w_{ij} positive) or inhibitory(w_{ij} negative) effect of u_j on u_i. Unit u_j has no direct connection to u_i if $w_{ij} = 0$. Figure 2.2 illustrates the relationship between the connectivity and the weight matrix.

In the general case it may be convenient to have separate groups or types of inputs. Connections arriving from neighboring



(a) A network with units ordered arbitrarily. The arrow on the connections indicate the sense of the connection. Numbers indicate weights on the connections.

$$W = \begin{array}{c} \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{array} \begin{array}{c} \begin{array}{ccccc} u_1 & u_2 & u_3 & u_4 & u_5 \end{array} \\ \begin{bmatrix} 0 & -7 & 0 & 0 & 0 \\ 7 & 0 & 9 & 0 & 0 \\ 0 & -9 & 0 & 0 & 0 \\ 10 & -8 & 3 & 0 & 0 \\ -3 & 8 & -10 & 0 & 0 \end{bmatrix} \end{array}$$

(b) The weight matrix for the network in (a).

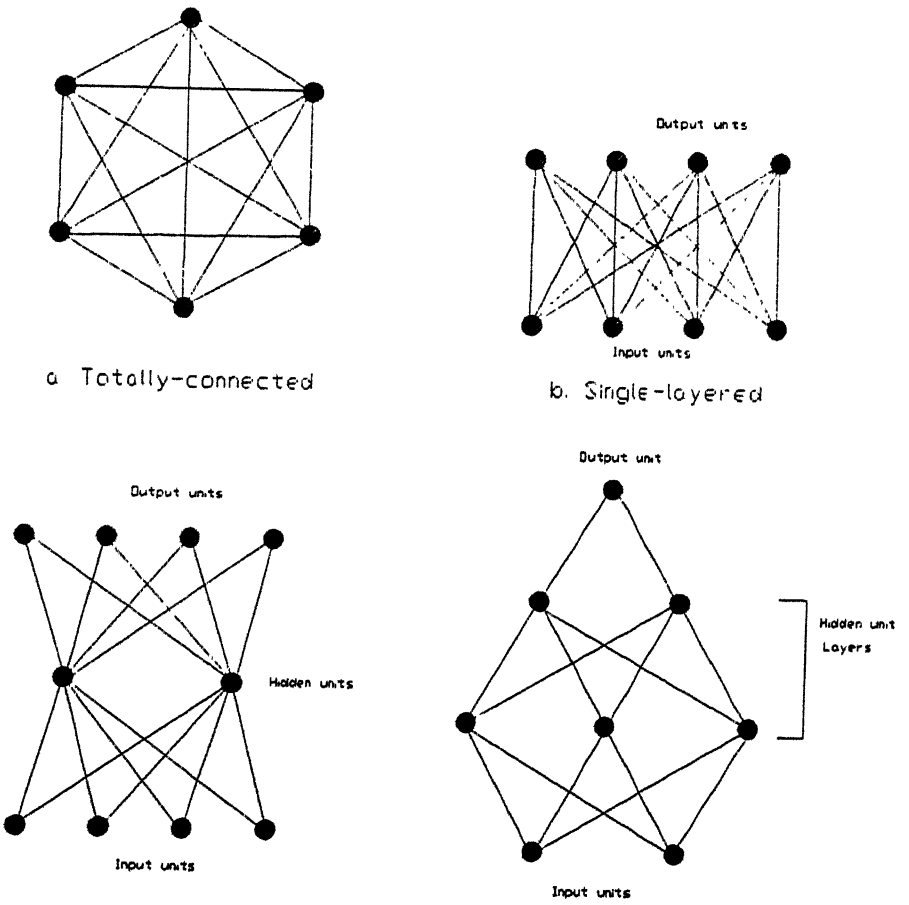
Figure 2.2 Connectivity and the weight matrix.

units may be visualized as arriving at reception centers or *sites*. Each site can be assumed to treat the incoming connections or links in a different way. Figure 2.1 shows units of two types

- A and B. Units of type A are shown to have a single site SØ with a single incoming connection. Units of type B have two sites, S1 and S2, at each unit. Site S1 receives inputs from two sources whereas, S2 has only one input. The connectivity pattern for such cases can be handled by having a separate connectivity matrix for each site. Thus we can represent the pattern of connectivity by a set of connectivity matrices, W_{io}, where i ranges over all sites.

Apart from connections between units, it is also possible to have connections between a unit and another connection - resulting in what are called *meta-connections*.

2.3.4 Network Architectures:



a. Totally-connected

b. Single-layered

c. Multi-layered

Figure 2.3 Network Architectures

In the most general case, every unit in the network is connected to every other unit. The resulting networks are *totally connected*. Usually, some connections are eliminated in order to improve performance and simplify learning. Some other architectures are shown in figure 2.3.

2.3.5 State of Activation:

For a system with N units, its *state* at time t is primarily specified by a vector (sometimes referred to as the state vector) of N real numbers, a(t), representing the pattern of activation over the set of processing units. The activation of unit u_i at time t is designated $a_i(t)$. Different models make different assumptions about the activation values a unit is allowed to take on. Activation values may be continuous or discrete, bounded or unbounded.

The state vector of the network represents the instantaneous *chunk of knowledge* currently being processed by the network. For a given architecture, portions of this vector may be used to represent the input and output of the problem.

2.3.6 Output of the Units:

Units interact by transmitting signals to their neighbors. The strength of their signals, and therefore the degree to which they affect their neighbors, is determined by their degree of activation. Associated with each unit, u_i, there is an output function, $f_i(a_i(t))$ which maps the current state of activation $a_i(t)$ to an output signal $o_i(t)$; that is

$$o_i(t) = f_i(a_i(t)) .$$

The current set of output values is the vector o(t). The function f may be the identity function, threshold function, some stochastic function, etc. Some authors do not distinguish between the activation of a unit and its output.

Units interact by transmitting signals to their neighbors. The strength of their signals, and therefore the degree to which they affect their neighbors, is determined by their degree of activation. Associated with each unit, u_i, there is an output function, $f_i(a_i(t))$ which maps the current state of activation $a_i(t)$ to an output signal $o_i(t)$; that is

$$o_i(t) = f_i(a_i(t)) .$$

The current set of output values is the vector o(t). The function f may be the identity function, threshold function, some stochastic function, etc. Some authors do not distinguish between the activation of a unit and its output.

2.3.7 Propagation Rule:

The rule of propagation takes the output vector, o(t), representing the output values of the units and combines it with the connectivity matrices to produce a net input for each site of the unit. We let net_{ji} be the net input of type i to unit u_j. If only one type of input (site) is used, net_j is used to mean the net input into unit u_j. In vector notation, $net_i(t)$ represents the net input vector for inputs of type i.

The propagation rule is generally straightforward; for units with all inputs belonging to the same type, we have:

$$net = W\, o(t) .$$

Different applications may require more complex rules of propagation.

2.3.8 Activation Rule

We also need a rule - the *activation rule* - whereby the net inputs of each type impinging on a particular unit are combined with one another and with the current state of the unit to

produce a new state of activation. This function, F, takes a(t) and the vectors net_j for each different type of connection and produces a new state of activation:

$$a(t+1) = F (a(t), net_1(t), net_2(t), ...) .$$

Again, F can be one of a variety of functions. Sometimes, we require F to be *differentiable*. One common class of activation functions is the *quasi-linear* activation function:

$$a_i(t+1) = F (net_i(t)) = F (\Sigma_j w_{ij} o_j).$$

2.3.9 Information Processing

There are two different methods for processing information in connectionist networks - *settling* and *feedforward processing*. Settling tends to be used primarily with auto-associative networks [see Chapter 4], and usually in the context of pattern completion or optimization tasks. Processing in these systems begins by encoding the input vector across the entire state vector. New activation states for the individual units are derived as a result of the propagation of signals between units. This iterative process of computing new states continues until the system "settles" into a stable state where no further changes are possible. At this point, the final state vector represents the output of the system. The updating of units may be done synchronously or asynchronously.

Feedforward processing occurs predominantly in multi-layered networks. The knowledge structure to be processed is first represented as an activity pattern across the vector of input units. This information is then propagated through the weighted connections to the next level of units. The resulting pattern of activity at this layer is, in turn, passed on to the next higher level. The feedforward processing of information continues in this step-wise, synchronous manner until a solution is represented as the activity across the final output layer. Feedforward

processing could be combined with *feedback*, in which case multiple iterations will be required.

2.3.10 Representation of the Environment

Typically, the environment is characterized by a stable probability distribution over the set of possible input patterns independent of past inputs and past responses of the system. If we list the set of possible inputs to the system and number them from 1 to M, the environment is then characterized by a set of probabilities, p_i for i = 1,...,M.

Since each input pattern can be considered a vector, it is sometimes useful to characterize those patterns with non-zero probabilities as constituting *orthogonal* or *linearly independent* sets of vectors.

2.4 Learning:

Learning in a connectionist network involves modifying the patterns of interconnectivity. In principle this can involve three kinds of modifications:

1. The development of new connections
2. The loss of existing connections
3. The modification of the strengths of connections that already exist.

(1) and (2) can be considered as a special case of (3). Changing a weight from a zero to a non-zero value has the same effect as growing a new connection. Similarly, changing the strength of a connection to zero is analogous to losing an existing connection. Moreover, very little work has been done on (1) and (2). Hence all learning procedures concentrate on rules whereby *strengths* of connections are modified through experience.

Several classes of learning rules are discussed below.

1. Variations on Hebbian Learning

Hebb's basic idea was that if a unit, u_i, receives a input

from another unit, u_j, then, if both are highly active, the weight, w_{ij}, from u_j to u_i should be *strengthened*. This idea has been extended and modified so that it can be more generally stated as

$$\Delta w_{ij} = g (a_i(t), t_i(t)) \; h (o_j(t), w_{ij})$$

where $t_i(t)$ is the teaching input to u_j and Δw_{ij} represents the change in the strength of the connection from u_j to u_i.

In the simplest versions of *Hebbian learning* there is no teacher and the functions g and h are simply proportional to their first arguments:

$$\Delta w_{ij} = \eta \, a_i \, o_j$$

where η is the constant of proportionality representing the learning rate. Another common variation is the *Widrow-Hoff rule* or *Delta rule* where the amount of learning is proportional to the difference between the actual activation achieved and the target activation provided by a teacher:

$$\Delta w_{ij} = \eta \, [t_i(t) - a_i(t)] \, o_j(t).$$

This is a generalization of the perceptron learning rule. Still another variation has

$$\Delta w_{ij} = \eta \, a_i(t) \, [o_j(t) - w_{ij}]$$

2. Unsupervised Learning

This form of learning involves self-organization in a completely unsupervised environment. Here again, many variations are possible; we shall restrict ourselves to the *competitive learning* scheme devised by Rumelhart and Zisper [1986]. The scheme is as follows:

* The units in a given layer are broken into a set of non-overlapping clusters. Each unit within a cluster inhibits every other unit within that cluster. The

clusters are *winner-take-all*, such that the unit receiving the largest input achieves its maximum value (say 1) while all the other units in the cluster are pushed to their minimum value (say Ø).

* Every element in every cluster receives inputs from the same lines.

* A unit learns if and only if it wins the competition with other units in the cluster.

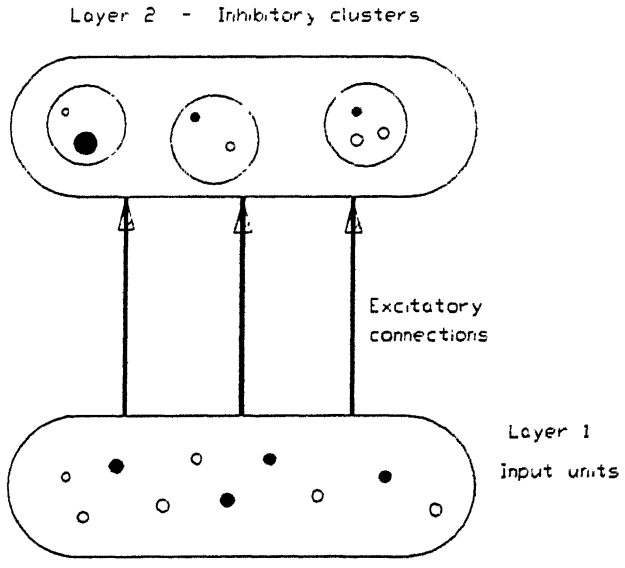


Figure 2.4
Competitive Learning.

* A stimulus pattern S_j consists of a binary pattern in which each element of the pattern is either active (1) or inactive (Ø).

* Each unit has a fixed amount of weight (all weights are positive) which is distributed among its input lines. The weight on the line connecting unit i on the lower layer (see figure 2.4) to unit j on the upper layer, is designated w_{ji}. The fixed total amount of weight for unit j is designated $\sum_i w_{ji} = 1$. A unit learns by shifting weight from its inactive to its active input lines. The learning rule can be expressed as:

$$\Delta w_{ji} = \begin{cases} 0 & \text{if unit } j \text{ loses on stimulus } k \\ g \dfrac{c_{ik}}{n_k} - g\, w_{ji} & \text{if unit } j \text{ wins on stimulus } k \end{cases}$$

where g is a proportionality constant, c_{ik} is equal to 1 if in stimulus S_k, unit i in the lower layer is active and zero otherwise, and n_k is the number of active units in pattern S_k (i.e. $n_k = \sum_i c_{ik}$).

3. Generalized Delta Rule

The delta rule works only for networks without hidden units. When hidden units are involved, the question of what are the "correct" activation values for the hidden units [i.e. $t_i(t)$ for hidden units] cannot be resolved: the external world does not specify what the correct state of a hidden unit should be.

The *generalized delta rule* (or the *back-propagation algorithm*), an extension of the delta rule, overcomes this problem and enables effective learning in multi-layered networks [Rumelhart, Hinton & Williams, 1986]. This rule works by the back-propagation of errors from the output units. The learning rule equations are:

$$\Delta_p w_{ji} = \eta\, \delta_{pj}\, o_{pi} \qquad (2.1)$$

where $\Delta_p w_{ji}$ is the change to be made to the weight from the i^{th} to j^{th} unit following presentation of pattern p, o_{pi} is the output of unit i on presentation of pattern p and η is the learning rate constant.

Two equations define the error signal δ_{pj}. For output units,

$$\delta_{pj} = (t_{pj} - o_{pj})\, f_j'(net_{pj}) \qquad (2.2)$$

where $f_j'(net_{pj})$ is the derivative of the activation function with respect to its total input, and t_{pj} is the target input to unit u_j

for pattern p. The error signal for hidden units (for which there is no specified target) is determined recursively in terms of the error signals of the units to which it directly connects and the weights of those connections. That is,

$$\delta_{pj} = f_j'(net_{pj}) \sum_k \delta_{pk} w_{kj} \qquad (2.3)$$

whenever the unit is not an output unit.

The most commonly used activation function, for units using the generalized delta rule for learning, is the logistic function:

$$o_{pj} = \frac{1}{1 + e^{-(\sum_i w_{ji} o_{pi} + \theta_j)}}$$

where θ_j is a bias similar to a threshold[1]. For this function,

$$f_j'(net_{pj}) = \frac{\partial o_{pj}}{\partial net_{pj}} = o_{pj}(1 - o_{pj})$$

with $net_{pj} = \sum_i w_{ji} o_{pi} + \theta_j$.

4. Meta-Generalized Delta Rule

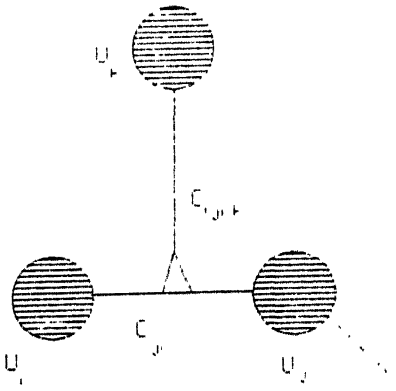


Figure 2.5
Meta-connections.

[1]Values of the bias θ_j can be learned just like any other weights. We simply imagine that θ_j is the weight from a unit that is always on.

The *meta-generalized delta rule* [Pomerleau, 1987] an extension of the generalized delta rule (GDR), uses the concept of a *meta-connection*. A meta-connection is a connection from a unit to the connection between two other units. In figure 2.5, there is a normal connection between u_i and u_j, denoted c_{ji}. There is also a meta-connection from u_k to c_{ji}. Such a meta-connection will be referenced as $c_{(ji)k}$. The *total weight* of c_{ji} on a given pattern presentation, denoted $W_{p(ji)}$ is just the normal weight of the connection, w_{ji} plus a non-decreasing function, f, of the net input to c_{ji} from meta-connections. Specifically,

$$W_{p(ji)} = w_{ji} + \sum_k w_{(ji)k} \sqrt{o_{pk}}$$

for all u_k with meta-connections to c_{ji}, where $w_{(ji)k}$ is the weight of the meta-connection $c_{(ji)k}$.

With this background, and with notation similar to that used for the generalized delta rule, the meta-generalized delta rule (MGDR) can be stated as follows:

$$\Delta_p w_{ji} = \mathcal{M} \delta_{p(ji)} \frac{o_{pi}}{net_{p(ji)}} \qquad (2.4)$$

and

$$\Delta_p w_{(ji)k} = \mathcal{M} \delta_{p(ji)} \frac{\sqrt{o_{pk}}}{net_{p(ji)}} \qquad (2.5)$$

where

$$net_{p(ji)} = o_{pi} + \sum_k o_{pk} \qquad (2.6)$$

$$\delta_{p(ji)} = \delta_{pj} o_{pi} \qquad (2.7)$$

When j is an output unit

$$\delta_{pj} = (t_{pj} - o_{pj}) f'_j (net_{pj}) \qquad (2.8)$$

When j is not an output unit

$$\delta_{pj} = f'_j(net_{pj}) \sum_k \delta_{pk} W_{p(ji)} \qquad (2.9)$$

with

$$net_{pj} = \sum_i W_{p(ji)} o_{pi} + \theta_j \qquad (2.10)$$

$$W_{p(ji)} = w_{ji} + \sum_k w_{(ji)k} \sqrt{o_{pk}} \qquad (2.11)$$

for all units k with meta-connections to c_{ji}.

Some explanation is in order. The delta values (equations (2.8) and (2.9)) are obtained in exactly the same way as for the generalized delta rule, except that the *total weight* $W_{p(ji)}$ is used in place of the normal weight w_{ji}. Even when computing net_{pj}, (equation (2.10)), normal weights have been replaced by total weights. $\delta_{p(ji)}$ can be thought of as the desired change in the total connection strength, $W_{p(ji)}$. This desired change in total connection strength is distributed to both the pure weight of c_{ji} and the meta-connections connecting to c_{ji} through equations (2.4) and (2.5). The quantity $net_{p(ji)}$ is a measure of the "total activation" projecting to a connection between two units, and it is used for dividing the total weight change between normal connections and meta-connections. $\mathcal{N}$ and $\mathcal{M}$ are learning rate constants for normal and meta-connections, respectively.

The activation function for the meta-generalized delta rule takes the form:

$$o_{pj} = \frac{1}{1 + e^{-(\sum_i W_{p(ji)} o_{pi} + \theta_j)}}$$

which is similar to the logistic activation function except that total weights are used here.

Though both the GDR and MGDR can be used with multi-layered networks, the MGDR has been shown to be superior in many ways.

5. The Boltzmann Machine

The Boltzmann machine [Hinton, Sejnowski & Ackley, 1984] is a *stochastic* connectionist network[2] in which the units have states of 1 or Ø, and the probability that unit j adopts the 1 state is given by

$$P_j = \frac{1}{1 + e^{-\Delta E_j / T}}$$

where $\Delta E_j = \sum_i s_i w_{ji} - \theta_j$ is the total input received by the j^{th} unit (also called the *energy gap*), s_i is the state of the i^{th} unit unit, w_{ji} is the strength of the connection from unit i to unit j, θ_j is the threshold[3] of unit j, and T is the "temperature". In this model, weights are assumed to be symmetrical, so that $w_{ij} = w_{ji}$.

Applying the above rule will make the network reach "thermal equilibrium". At thermal equilibrium, the units still change state, but the probability of finding the network in any global state remains constant and obeys a Boltzmann distribution in which the probability ratio of any two global states depends solely on the energy difference:

$$\frac{P_a}{P_b} = e^{-(E_a - E_b) / T}$$

The global energy function is defined by

$$E = -\frac{1}{2} \sum_{i,j} s_i s_j w_{ij} + \sum_i s_i \theta_i$$

---

[2] The Boltzmann machine is actually a generalisation of the Hopfield network. For details of the Hopfield network, see chapter 4.

[3] The threshold term can be eliminated by giving every unit an extra input connection whose activity level is fixed at 1. The weight on this special connection is the negative of the threshold, and it can be learned in just the same way as the other weights.

where the symbols have their usual meanings.

At high temperature, the network reaches equilibrium rapidly but low energy states are not much more probable than high energy states. At low temperatures the network approaches equilibrium slowly, but low energy states are much more probable than high energy states. The fastest way to approach low-temperature equilibrium is to use *simulated annealing* - start at a high temperature and gradually reduce the temperature. Simulated annealing allows Boltzmann machines to find the low energy states with high probability. Thus, if some units are clamped to represent an input vector, and if the weights represent constraints of the task domain, the network can settle on a very plausible output vector given the current weights and the current input vector. The Boltzmann machine is therefore a constraint satisfaction network.

Learning Procedure: The Boltzmann machine learning algorithm has two stages. The network is first "shown" the mapping it is required to perform by clamping an input vector on the input units and clamping the required output vector on the output units. If there are several possible output vectors for a given input vector, each of the possibilities is clamped on the output units with the appropriate probability. The network is then annealed until it approaches thermal equilibrium at a temperature of 1. It then runs for a fixed time at equilibrium and each connection measures the fraction of the time during which both the units it connects are active. This is repeated for all the various input-output pairs so that each connection can measure $\langle s_i s_j \rangle^+$, the expected probability, averaged over all cases, that unit i and unit j are simultaneously active at thermal equilibrium when the input and output vectors are both clamped.

The second stage involves running the network in the same way but without clamping the output units. Again, it reaches thermal equilibrium with each input vector clamped and then runs for a fixed additional time to measure $\langle s_i s_j \rangle^-$, the expected probability

that both units are active at thermal equilibrium when the output vector is not clamped. Each weight is then updated by an amount proportional to the difference between these two quantities:

$$\Delta w_{ji} = \varepsilon (\langle s_i s_j \rangle^{+} - \langle s_i s_j \rangle^{-})$$

The Boltzmann machine learning rule provides yet another method for learning the weights of hidden units.

In summary, this chapter gives a general framework for looking at connectionist models along with a somewhat detailed exposition of a variety of learning algorithms. A comparison of the various architectures and learning rules, stressing their plus points and drawbacks has, however, not been provided. Performance analyses can be found in any of the references quoted.

CHAPTER 3

CONNECTSIM++ : THE NITTY-GRITTY

3.1 Overview:

This chapter begins with an outline of the design philosophy of ConnectSim++. The organization of the system is then described along with the main system objects - units, sites and links. In the course of this, the user gets introduced to the terminology used in the rest of the thesis. The conceptual network structure as modeled by the system, and its physical representation in the computer's memory is then explained.Following this, the different phases of network construction and simulation, together with the operations involved in each phase and the system primitives to implement them are explained with illustrations of their use.

The main objective is to give the user a feel for the use and flexibility of the various features that the system provides; not all of the functions and commands supported by the system have been explained. The complete set of commands along with the details of their syntax and operation has been relegated to the ConnectSim++ Users' Manual in Appendix A.

The sequence of the tasks involved, and the use of the system-provided primitives to implement them, is made clearer with an illustrative example, wherein a simple network is constructed and the simulation carried out. Finally, some salient features of the system have been briefly explained.

3.2 The Design Philosophy:

Connectionist network research is an area which is still in a state of flux. A unified theory that describes a network with

capabilities that encompass all the goals set for artificial neural nets - in other words, whose performance approaches anywhere close to that of the animal brain - is yet to emerge. Right now, there exist a variety of models differing in complexity, with their own idiosyncrasies, and each one suitable for a particular class of tasks. They consist of units with varying characteristics and structures,interconnected in specific patterns of connectivity and use different rules for learning.

ConnectSim++ is an attempt at developing a simulation tool that is adaptable to the changing requirements in this complex scenario. The underlying objective in the design of the simulator has been flexibility and versatility. Steps that are common to the specification and simulation of all networks (as dictated by the general framework for connectionist models described in the previous chapter) have been identified and primitives that implement them have been provided. Facilities have been provided to the user to specify features that are peculiar to each network. Examples of such features are the method of obtaining the input data, the unit activation functions, the learning rule for the network etc. Such functions are specified in either C(ANSI standard) or C++. The more common among such functions are directly callable from a library. The user can enhance the library by adding his own functions to it.

The system itself has been implemented in C++ to exploit the advantages offered by an object-oriented approach. This has enabled the system objects to be neatly packaged in a structure that contains data representing the object, as well as functions for their manipulation.The overall result is a neat and elegant system that can be easily modified/extended. (In some places, however, adherence to the principles of object-oriented programming had to be relaxed in favor of efficiency.)

One of the important measures of the "goodness" of such a system is its user-friendliness. The attempt has been to make the user-interface as simple, yet powerful, as possible. A variety of facilities for examining the network as it is being constructed, as well as during simulation, are made available, along with features that enable the user to dynamically control the information that is displayed, as well as its detail. A facility to set a break in the simulation to facilitate debugging and a closer observation of the network is supported.

3.3 System Organization:

The network model supported by the system conforms to that described in the previous chapter: Any network is assumed to be made up of units interconnected in particular ways. Each unit can have a number of sites at which connections or links from neighboring units arrive. All connections arriving at a particular site are handled in a way that is characteristic of that site. The activation or potential of any unit decides its contribution to the overall network behavior and, generally, is some function of its activation during the previous time-step and the current inputs. The system associates a function - the activation function - with each unit, site and link, which represents its action in the network. Conceptually, such a network can be depicted as shown in Fig. 3.1(a).

Memory Representation:

The choice of the data structures for the network were guided mainly by considerations of time and space efficiency, flexibility and self-sufficiency. The motivation for the last consideration comes from the fact that all models proposed for the neural units have implied that all the information required for the computations performed by a unit, are available locally at the unit. The C++ class ideally fits these requirements.

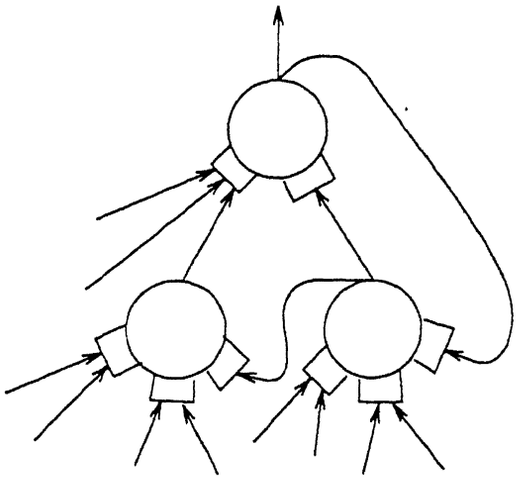


Figure 3.1 a Conceptual Representation of the Network Model

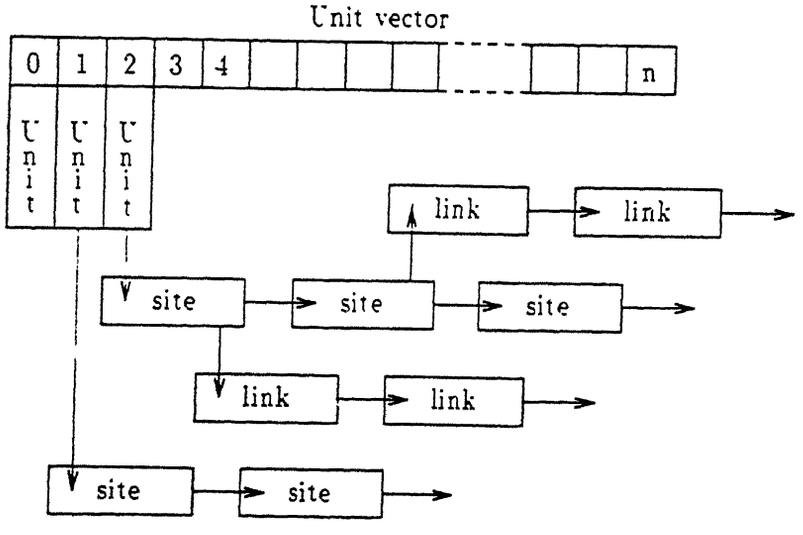


Figure 3.1 b Memory Representation of the Network Structure

This choice yields the benefits of modularity, controlled access to data and scope for future enhancement. Any enhancement in the functionality of an object is easily achieved by defining a derived class of the corresponding class, with the desired extra features. Also, such an enhancement has little or no effect on the rest of the code.

The main data structure is a one-dimensional array of *Unit* objects (of class *Unit*). The sites present at each of the units are represented by a linked list of *Site* objects (of class *Site*). A linked list of *Link* objects (of class *Link*) represents the incoming connections to a site. This scheme is depicted in Fig. 3.1(b).

3.4 System Objects:

(a) *Units*: Every unit in the system is associated with several pieces of data such as the potential, output and state. The potential corresponds to the unit's level of activation. The output represents the value that is transmitted from the units along the connections leaving the unit and is generally some function of the potential. The state is a small integer that can be used to make simple decisions about how to interpret the unit. Associated with each unit is a Unit Activation Function, which determines the unit's potential, its output and state from the net inputs of the attached sites. A fan-out list gives a list of all unit-site pairs which receive input from the unit.In addition, each unit also stores other information such as the name of the unit, its type, its set membership and flags. This information helps either in presenting the user a more friendly interface or in implementing the actions of the simulator more efficiently.Each unit in the network is referenced by its index into the unit array,or a pointer to the unit. However, at a higher level, individual units can also be named and subsequently referenced by that name. Pointer

referencing is the fastest and is the preferred access method, especially in tight loops within functions.

(b) *Sites*: These act as input collection points to the units. There can be zero or more sites attached to a unit with each one of them receiving zero or more inputs from other units. (There can also be links which feedback an unit's output to itself.) Each site has a name and, together with the specification of the unit to which it is attached, provides a means of accessing it uniquely. A site activation function associated with the site computes the net input from all the connections coming into the site.

(c) *Links*: These connect the output of an unit to a particular site of another(sometimes the same) unit. Each link object stores information about the weight and the output of the unit where the link originates. Since a link is attached to a site at the destination unit, it also stores the unit identification (index) of the source unit. A link is uniquely identified by specifying the source unit, the destination unit and the site where it arrives at the destination unit.

From within functions, each of the objects is also accessible through pointers to them. All of the above system objects have, in addition, a general-purpose data field, and several member functions which implement common functions involving the particular object such as initialization, display, addition of sites to a unit and so on. The unit potential and output, site and link value fields, link weights and the unit, site and link data fields can store either integer or floating point values. (The simulator handles this in a special way, as will be explained in the final section of this chapter.)

The Users' Manual contains more information on the contents of the *Unit*, *Site* and *Link* classes and their member functions.

3.5 Activation Functions and Updating the Units:

During simulation the units are updated by first invoking the link activation functions for all the links arriving at that unit; the site activation function is then used for all the sites attached to the unit to compute the net input at those sites by combining the weights of the incoming connections with the output value of the unit at the other end of the connection. Finally, the call to the unit activation function updates the potential, output and the state of the unit. Since the activation functions are user-written, the user has complete flexibility in deciding what shall be done in each of these functions, though very often they are restricted to the stated purpose of performing actions associated with the corresponding unit, site or link.

These functions may be one of the library-provided functions or a function written by the user in C(ANSI standard) or C++ and compiled with the simulator it is the case that there is no action associated with a unit, site or link,a null function (i.e., which does nothing) is specified for the corresponding activation function. When a unit has a null function or when it is required to disable a non-null activation function associated with it, the *NO_UNITFUNC_FLAG* of that unit is set. This results in the unit activation function for the unit not being called in the subsequent steps of the simulation i.e., the unit is not updated. Also, if a unit has no sites attached or has sites all of which have null site activation functions or if it is required to temporarily disable the site activation function, the *NO_SITEFUNC_FLAG* may be set, to inform the simulation system that the site activation function for any of the sites attached to the unit need not be invoked during simulation. The *NO_LINKFUNC_FLAG* of a unit may be used to a similar purpose with the links arriving at the unit.

In the case where a unit has a null activation function, or has sites or links all of which have null activation functions,

the ultimate result is the same whether or not the corresponding flag is set. However, it is always advisable to set the flag, since this results in the simulation being run more efficiently. Note that setting the *NO_SITEFUNC_FLAG* or the *NO_LINKFUNC_FLAG* of a unit affects *all* sites or links at the unit.

3.6 The Simulation Process:

The process of simulating any given network consists of (assuming that the network is already constructed) the following sequence of operations:

(i) clamping the input vector on the set of input units and the desired output vector (target vector) on the teacher units, if they exist.

(ii) updating each of the units, sites and links in the network as per the activation functions associated with them (as described above).

(iii) applying the learning rule to update the weights of the links.

The entire sequence (i) through (iii) described above constitutes a *step*. Within each *step*, the action specified by (ii) may be performed several times. Each execution of (ii) is said to constitute an *iteration*. The entire simulation process may be spread over several steps, each consisting of a given number of iterations.

To provide maximum flexibility in carrying out a simulation, each of the actions above uses functions specified by the user - either his own or those from the library.

The network can then be tested to see how effectively it has learnt by applying each of the input vectors to the input units, updating all the units by invoking their respective activation functions a required number of times and verifying that it reproduces on the output units the patterns that it was taught.

Of course, during testing the teaching inputs are not present, and the learning rule is not applied.

This is most commonly the structure of the simulation process. In addition to the above, certain other actions may be specified which allow the user to follow the dynamic behavior of the network during simulation or to report the progress of the simulation process.

The simulation can be performed in one of two modes - synchronous or asynchronous:

(i) If synchronous simulation is specified, during each iteration all the units are updated using weights and unit output values as computed during the previous step. The network behaves as though all the units update simultaneously.

(ii) If asynchronous mode of simulation is chosen, during each iteration the units are updated in pseudo-random order and the new output value is immediately transmitted to the neighboring units.

Synchronous simulation is marginally more efficient than asynchronous simulation. However, in some cases it becomes mandatory to use asynchronous mode of simulation, since synchronous simulation could lead to oscillations, other problems of failing to break symmetry or can always lead to the same outcome.

3.7 Phases in Network Construction and Simulation:

The process of constructing a network, performing the simulation and testing the network involves several distinct steps and proceeds in clearly distinguishable phases. The tasks involved in the specification and simulation of a network, (in the order in which they are typically performed) may be broadly identified as:

(i) Specification of the user-written functions for network construction,data input (if any), unit, site and link activation functions (if the library functions are not usable) and the network learning rule, and then compiling these with the simulator code. The user may create files containing these functions and compile them with the system using the *simgen* command with appropriate options for creating the integer or floating point version of the simulator.

(ii) Start up the simulator by typing the command *sim* against the shell prompt of the system.

(iii) Construct the network by specifying the units, attaching sites to them and setting up connections between the units. Unless the structure of the network is very simple, this should preferably be done with a function specified by the user and already compiled with the simulator.Such a function can be called from the command interface for building the network. The user-written functions for this purpose make use of appropriate library functions and are generally quite straightforward to write. This is strongly recommended (and in many cases inevitable) for its efficiency and elegance. Constructing the network entirely from the command interface can prove to be quite awkward for want of looping constructs in the simple command language.

(iv) Specify the network input and learning functions, which can be ones from the library or those provided by the user.

(v) Indicate the units, sites and links to be displayed during simulation. Also certain other display parameters which control the amount of information that is displayed can be set up. If needed, a break in the simulation after a certain number of steps can be specified to facilitate debugging the network.

(vi) The simulation is then started by indicating the number of steps the simulation should run and the number of iterations per step. (Alternatively, a limiting value for

the iteration error can be specified; a number of iterations sufficient to reduce the error to below this specified value are performed in each step.)

(vii) When the simulation is complete, the network is tested for its performance. The user may also output any results by invoking an appropriate function.

(viii) Before finally exiting the simulator, the file containing the log of the commands executed during the session can be saved, so that the next time the entire sequence of commands need not be retyped to repeat the same set of operations; the commands can be simply read from the log file.

3.8 The Programming Interface and the Command Interface:

The simulator system provides primitives for the performance of the individual operations involved, via two interfaces:

(i) a *Programming Interface* or a *Function Call Interface.*
(ii) a *Command Interface.*

In the discussion that follows, for the purpose of distinguishing the primitives available from the two interfaces, the primitives accessible through the programming interface will be referred to as *functions* while those accessible through the command interface will be referred to as *commands.*

Both interfaces make available equivalent primitives for performing the individual subtasks of network specification and simulation. Thus,theoretically, it is possible to carry out the entire process using the facilities provided at any one interface. However, this is not always recommended, since it does not allow the completion of the task in the most effective and efficient manner. Depending on the nature of the problem being simulated and the relative complexity of the subtasks, (for example, the complexity involved in specifying the network and certain other

considerations such as whether library-provided functions are being used as activation functions and the form in which the final result is required to be displayed) it is advisable, to use a mixture of primitives from the two interfaces that enables performing the task in the most optimum manner.In designing the simulator care has been taken to see that the granularity of the operations performed in various steps is such that the user is provided utmost ease and flexibility, while retaining complete control over the process.

The paragraphs that follow describe the set of operations that are commonly used to construct a network and simulate it. As mentioned above, though both functions and commands exist that implement the same operation, we describe the more commonly used of the two for the purpose. Of course, the details about the other equivalent primitive for the same can be found in the Users' Manual, along with the rest of the commands and functions.

3.9 Network Construction:

Essentially, the network construction process involves creating the units of the required type, attaching sites to these units and setting up connections between units. The parameters that are associated with units,sites and links are initialized at the time of their creation. However,these can be modified at a later stage in the network construction process and during simulation. Also, it is possible to name the units and group them into sets to impart some kind of a logical structure to the network,and for the sake of convenience in terms of accessing and displaying these.The system supports all of these operations.(In the following paragraphs, the function headers shown conform to the C++or ANSI C syntax, with the function type as well as the type of the parameters explicitly specified in the function header/declaration.)

3.9.1 Creating Space for the Units:

Before the network construction can begin and any units are made for the purpose, space should be created for the anticipated number of units that exist in the network. This is achieved by a call to the *AllocateUnits* function:

void AllocateUnits(int number);

This not only creates space for the specified number of units, but also creates an Outputs array (used to store the output values of the units during simulation) of a corresponding size.

Generally, the number of units required for the network will be known in advance, and a single call at the start of the network construction to *AllocateUnits* creates all the space required for the units. However, it is perfectly valid to invoke the function at any stage of the network construction process to create extra units. Such a call would return a larger unit array with the previously created units already in it and containing enough space for the extra units. In fact, an attempt to create more units than the number specified to a prior call to Allocate units,automatically creates space for a prespecified number of extra units.

3.9.2 Making Units:

The *MakeUnit* function creates an unit of the type specified and with the specified parameters, using the space allocated by *AllocateUnits*:

*int MakeUnit(char *type, int init_state, int state, int init_pot, int pot, int data, ufunc_ptr UnitFunc, int output);*

where the arguments specify the values for the corresponding unit parameters. *UnitFunc* is one of the library functions or a

user-specified function or can be specified as NULL, in which case the activation function is taken to be the null function. *MakeUnit* returns the index in the unit array, of the unit created by the call. The first call to *Makeunit* returns index Ø and subsequent calls return consecutive indices.

A typical call to *MakeUnit* would look like:

```
int unit_index = MakeUnit("Auto",1,1,50,50,0,UFauto,0);
```

3.9.3 Adding Sites:

One or more sites can be added to an unit by calls to the function *MakeSite*:

```
Site *MakeSite(int u_index, char *name, sfunc_ptr SiteFunc,
int data);
```

where u_index is the index of the unit to which a site indicated by *name* is being attached. *SiteFunc* gives the pointer to the site activation function which can be user-specified, a library- or a null function. *Data* gives the value with which to initialize the data field of the site.The call returns a pointer to the newly attached site. Example:

```
Site *sp = MakeSite(i, "feed", SFweightedSum, 0);
```

3.9.4 Making Connections:

Connections are set up between units with the *MakeLink* function:

```
Link *MakeLink(int from, int to, char *site_name, int weight,
                                 int data,   lfunc_ptr LinkFunc);
```

where *from* and *to* indicate the indices of the source and

destination units respectively, *site_name* gives the name of the site at which the connection is made at the destination unit. *weight* and *data* specify the initial values for the weight of the connection and the link data field respectively. *LinkFunc* is a pointer to the link activation function. The function returns a pointer to the link that is set up.

Example:

MakeLink(i, j, "teach", -50, 0, NULL);

Such a call creates a connection from unit *i* to the site *teach* of unit *j* with the weight of the connection being -5Ø. The *data* field is initialised to Ø and the pointer to the link activation function is set to NULL.

3.9.5 Deleting Sites and Links:

Functions for deleting sites attached to units and connections between units are the *DeleteSite* and *DeleteLink* respectively. However, these functions are rarely used during network construction.

3.9.6 Naming:

The system maintains a *Name Table* to store and retrieve names for units,unit types, sites, sets, functions and states. Information related to each of the names are also stored in the name table. Functions for directly accessing the name table are seldom used and are described in the Users' Manual.

Naming Units:

Internally, the system never has any use for unit names. However,appropriately naming units or groups of units will serve as a mnemonic aid to the user in accessing units, or in imparting a logical structure (that corresponds to the network being

constructed) to the unstructured collection of units. Generally, it is convenient for the user to access units by their names rather than by their indices, from the command interface during network construction and simulation.

The function *NameUnit* enables naming of single units or a vector or a 2-D array of units:

*void NameUnit(char *name, int type, int index, int cols, int rows);*

name is the pointer to the character string giving the name and *type* specifies the type of the name - *SCALAR* (for a single unit), *VECTOR* (for a 1-D array or vector of units) or *ARRAY* (for a 2-D array of units) (not to be confused with the unit type). *Index* gives the index, in the unit array, of the unit to be named or of the first unit in the vector or array.The argument *cols* need be specified only if the type is *VECTOR* or *ARRAY*. Similarly, *row* is specified only if the type is *ARRAY*.

Subsequently, the units may be accessed by their names. The name has the form *name[c]* if the name is of type *VECTOR* and *name[r][c]* if it is of type *ARRAY*, where *c* and *r* denote the indices. *cols (rows * cols)* consecutive units are given the specified name if the name is of type *VECTOR (ARRAY)*.

Example:

NameUnit("Font", ARRAY, 20, 5, 8);

assigns the name *Font* to 40 units (units with indices 20 through 59) in the unit array, creating a logical 2-D unit array of 8 rows of 5 units. Now the unit whose index in the unit array is 36 is accessible as *Font[3][1]*.

Naming States:

The state associated with each unit in the system is a short integer useful in making simple decisions on how to interpret the unit. However, it would be much clearer, if a state were displayed as a name rather than a number. The Declare state function associates a given name with a state. For example,

DeclareState("Dead", -1)

will cause *"Dead"* to be displayed in place of -1 for the state of the unit.

3.9.7 Unit Sets and Set Operations:

During network construction and simulation, sometimes need will be felt for specifying an operation such as addition of sites or display, on an arbitrary collection of units which share some common feature. For this purpose the simulator system allows creation of unit sets and manipulation of these sets. Sets are referenced by their name and are especially useful for specifying operations from the command interface.

Functions/commands are provided for all essential set-theoretic operations:

- *DeclareSet(char *name)* - creates a set, which is initially empty, with the specified name.
- *AddToSet(char *name, int low, int high)* - adds units with indices *low* through *high* to the given set.
- *RemFromSet(char *name, int low, int high)* - removes units with indices *low* through *high* from the given set.
- *UnionSet(char *name3, char *name1, char *name2)* - assigns the union of sets *name1* and *name2* to the set *name3*. Creates set *name3* if it does not exist.
- *IntersectSet(char *name3, char *name1, char *name2)* -

assigns the intersection of sets *name1* and *name2* to the set *name3*. Creates set *name3* if it does not exist.

- *DifferenceSet(char *name3, char *name1, char *name2)* - assigns the difference of sets *name1* and *name2* to the set *name3*. Creates set *name3* if it does not exist.
- *InverseSet(char *name2, char *name1)* - assigns to set *name2* all units not in set *name1*. Creates set *name2* if it does not exist.
- *DeleteSet(char *name)* - deletes the set with the specified name.
- *MemberSet(char *name, int unit_index)* - returns *TRUE* if the unit given by *unit_index* is a member of the set *name*,else returns *FALSE*
- *MemberOfSets(int unit_index)* - displays the names of all sets of which the unit given by *unit_index* is a member.

3.9.8 Modifying and Accessing Unit, Site, and Link Parameters:

The *Unit*, *Site* and *Link* classes have inline member functions which enable the user to access and modify values of the parameters (i.e. potential,output, weight, data etc.) in the private section of the class. These member functions can be invoked with the 'dot' (.) or the 'arrow' (->) operators used for accessing members of a class ("C++ style") from within C++functions. Alternatively, "C style" functions have been provided which take the unit index as a parameter and using that invoke the member function of that unit object to access the desired parameter. Similarly, for sites the functions take as parameters the unit index and site name (and for links the indices of the source and destination units and the destination site), and use them to invoke the corresponding member function. Equivalent functions which access these objects through pointers are also provided. These can be used from within C(ANSI standard) or C++ functions. These functions are described in the Users' Manual.

3.10 Examining the Network:

The system provides functions and commands to display information about units, sites and links. These help in verifying that the network has been constructed correctly and in finding out the values of the various fields. Information about units and sites can be displayed in detail so that all aspects of the network can be examined. It is also possible to have the units and sites displayed compactly, with only the important fields being displayed. The commands and functions allow a range of units or a set of units to be specified for displaying. To distinguish the commands(functions) which provide detailed information from those that output information in compact form, the former are referred to as *Display* commands (functions) and the latter as *List* commands (functions). (This remark does not apply to the case of the links, however, since not much information is present at a link.)

- *DisplayUnit(int low, int high)*; - Displays the units with indices in the range *low* through *high*.
- *ListUnit(int low, int high)*; - Displays the units with indices in the range *low* through *high* in compact form.
- *DisplayUnitSet(char *name)*; - Displays the units which are members of the set specified by *name*.
- *ListUnitSet(char *name)*; - Displays the units which are members of the set specified by *name* in compact form.
- *DisplaySite(int low, int high, char *name)*; - Displays the sites with name specified by *name* and attached to units with indices in the range *low* through *high*. If a site of that name does not exist at some unit in the specified range, the function reports so, if the *verbose* mode is set and is silent otherwise.
- *ListSite(int low, int high, char *name)*; - Displays the sites with name specified by *name* and attached to units with indices in the range *low* through *high* in compact form. If a site of that name does not exist at some unit in the specified range, the function reports so, if the *verbose*

mode is set and is silent otherwise.

- *DisplayLink(int from, int to, char *name)*; - Displays the link connecting the unit with index *from* to the site *name* at the unit with index *to*.
- *ListLinksTo(int low, int high, char *name)*; - Displays all the links arriving at the site *name* of the units with indices in the range *low* through *high*. If a site of that name does not exist at some unit in the specified range, the function reports so, if the *verbose* mode is set and is silent otherwise.
- *ListLinksFrom(int low, int high)*; - Displays all the links originating at the units with indices in the range *low* through *high*.

Corresponding commands also exist and are described in the Users' Manual.In fact, these allow more flexible specification of units - by their indices, by their names or by the reserved word *all*. The reserved word *all* is allowed in place of a site name as well.

Besides these, facilities are provided which enable the display of units to be dynamically controlled during simulation: At the end of each step of the simulation, units which have their *SHOW_FLAG* set to 1 and those which are members of the ShowSet are displayed in detail. *SHOW_FLAG* is one of the user-settable flags present at each unit.ShowSet is a set which is created by the system at start up and is initially empty. One of the user-written functions (most commonly the unit activation function) can switch on the *SHOW_FLAG* or add an unit to the ShowSet, based on some criterion to have the unit displayed during subsequent steps of the simulation. When an unit is to be deselected for displaying, the function can remove it from the ShowSet (if it is a member) or switch off the *SHOW_FLAG* (if it is on). Also, during simulation, the units with their *LIST_FLAG* on will be displayed in compact form. Activation functions can turn this flag on or off, as with the *SHOW_FLAG*, to control the display of the associated object.

In addition, there are functions and commands which enable the display to be passed through a pipe (e.g. /usr/ucb/more to avoid the display scrolling off the screen) or to pause the display after each show or listing. All these provide the user with a lot of flexibility and control.

3.11 Simulation of the Network:

Prior to starting the simulation the user has to specify the functions to be used by the network for obtaining data for the input units and clamping them on the input units (the input function), and for updating the link weights (the learning algorithm). The network input function (if one exists)has to be specified by the user. The learning rule can, however, be chosen from the library. By default, these are taken to be the null function. When the simulation is started, if these are still set to the null function, the system reports this fact to the user. These functions are invoked once in every step of the simulation.

As explained in 3.5, simulation can be performed in one of two modes -synchronous or asynchronous. If the simulation is required to be asynchronous, the user has to indicate this to the system. The default simulation mode is synchronous. Simulation is generally run from the command interface since this allows a lot of flexibility in setting parameters which control the display.

The network input function is set using the *setinputfunc* command:

setinputfunc *<name>* *<RETURN>*

sets the input function to the function specified by *<name>*. (All arguments to a command which need to be specified by the user will be shown enclosed in < >. The *<RETURN>* at the end indicates that the *Return* key of the keyboard is to be pressed. The command itself is typed against the simulator prompt, ->.)

The network learn function is set using the *setlearnfunc* command:

setlearnfunc <name> <RETURN>

sets the function specifying the learning rule to be the function specified by *name*.

The simulation is (explicitly) specified as synchronous with the *sync* command whose syntax is simply

sync <RETURN>

The simulation can be specified to be asynchronous with:

async <RETURN> or *async <seed> <RETURN>*

where *<seed>* is an integer with which the random number generator is to be seeded. (Remember that during asynchronous simulation the units are updated in a pseudo-random order ?)

These commands only specify the mode of the simulation and do not actually cause the simulation to be run. Before the simulation is started the user can specify any units that need be *shown* or *listed* using the *show* and *list* commands respectively. Other flags which control the display such as *pipe* and *pause* can be set. The user can also get a message of the number of steps completed by setting the *Echo* flag with the *echo* command.

In addition, a *break* in the simulation can be indicated with the *break* command. This causes the simulation to be suspended after a certain number of steps and return control to the user, whereupon he can execute any command to examine the network or even to change some network or system parameter. A set of commands may also be specified for automatic execution upon breaking. When the user is through with doing whatever he intends to,he can resume the simulation with the *continue* command.

Finally, the actual simulation can be commenced with the *simulate* command:

simulate <steps> <iterations> <RETURN>

where *<steps>* and *<iterations>* indicate the number of steps of simulation to run and the the number of iterations in each step.

Sometimes, it may be required to repeat the iterations until the iteration error (set by one of the activation functions) has reduced to below a given value. This is done with:

simulate <steps> with itererr <error> <RETURN>

where *<error>* indicates the limiting value for the iteration error. The iteration within a step is terminated when the iteration error as computed by the activation function is less than or equal to the value specified by *<error>*.

When the indicated number of steps have been completed, the simulation ends with a message indicating the time taken to complete the simulation.(This is only approximate and also includes the time taken up in the display and pause actions.) The network weights would have been updated to reflect the learning undergone by the network.

The network may then be tested for its performance by running the *test* command in either of its two forms:

test <steps> <iterations> <RETURN>

test <steps> with itererr <error> <RETURN>

where *<steps>*,*<iterations>* and *<error>* have the same meaning as with the *simulate* command. The *test* command sets the inputs to the network and causes the corresponding outputs to develop on the output units (assuming that the learning has taken place

correctly) by updating the units, sites and links in the network. However, the target inputs (if any exist) are not set and the learning rule is not applied at the end of each step.

The above discussion describes the essential functions and commands that enable a network to be constructed and simulated. The following section presents an example in which a simple pattern associator network is specified and simulated, to illustrate the use of these functions and commands.

3.12 Example - A Pattern Associator Network:

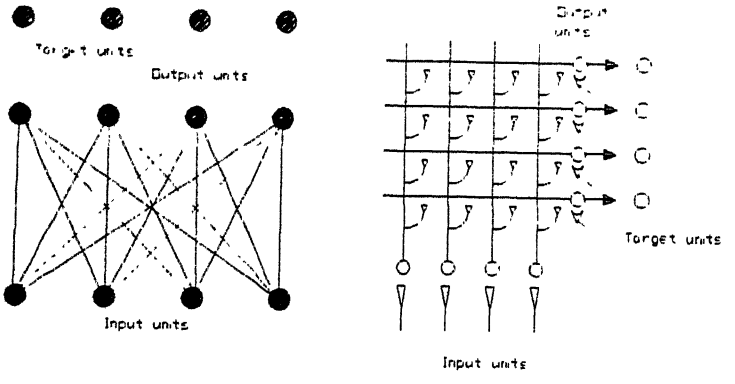


Figure 3.2 Pattern Associator.

A pattern associator is a network in which a pattern of activation over one set of units causes a pattern of activation over another set of units.We shall consider a model which has four input and four output units. Each input unit is connected to each of the output units. Figure 3.2 shows two ways of representing the network.

Updating Rules: The activation of the input units is decided by factors external to the network. In this case, the input pattern presented from outside the network is used to set the outputs of the input units.

For the output units, we have

$$net_j = \sum_i (w_{ji} * o_i)$$

where net_j is the total input to unit j, o_i the output of input unit i, w_{ji} is the weight of the connection from u_i to u_j and i varies over all input units. The output of u_j is:

$$o_j = \begin{cases} 1 & net_j > \emptyset \\ \emptyset & net_j \leq \emptyset \end{cases}$$

The output for all units in the network is either Ø or 1.

Input and Output Patterns:

Let us assume that the pattern associator is required to learn the following mapping:

	Input presented				Output required			
	u_1	u_2	u_3	u_4	u_1	u_2	u_3	u_4
1	Ø	1	1	Ø	1	Ø	Ø	1
2	1	1	1	1	Ø	Ø	Ø	Ø
3	Ø	1	Ø	1	1	Ø	1	Ø

Learning Rule: On a learning trial, the input pattern is clamped on the input units and the target units (figure 3.2) provide a teaching input to each output unit, specifying what its value ought to be. The pattern associator now computes the output for the given pattern. For each output unit, the network compares its answer with the target. Weight changes are calculated as follows:

- If the actual output matches the target output, no changes are made.
- When the computed output is Ø and the target says it should be 1, increase the weights of the connections from all active input units (with output = 1) by a small amount n.
- When the computed output is 1 and the target specifies Ø,

the weights from all the input units that are active are reduced by n.

This rule is actually the *perceptron convergence procedure* (Section 2.2(1)).

Network, Definition and Simulation: Before we can actually construct the network and perform the simulation, we have to specify the functions for building the network, reading the input patterns, the learning rule and for updation of the units (i.e., activation functions for units, sites and links). The file containing these functions is shown in Fig. 3.3 (The comments provided make the functions self-explanatory.)

```
/* Program for simulation of a Pattern Associator Network.
Comments are preceded by "//" as in C++. */
// Files containing the definition of the simulator
#include "defs.h"
//    data structures and other global variables.
#include "extern.h"
#include "sim.h"
// Constant representing the learning rate.
#define ETA 10
typedef int * intp;
// Array to store the input and target patterns.
intp *ipat, *tpat;
// Library function for net site input.
func_type SFweightedSum(Site *);
FILE *infile;
// No. of input (also output & target) units
int unitCount;
// No. of patterns.
int patCount;
```

Figure 3.3 (contd.)

```
/* Function for constructing the network. */
void BuildNet(){
int i, j;
     if (argc !=3){   /* Check the number of arguments */
          printf("Usage: call BuildNet <file_name>");
          return;
     }

// Open the input data file; the name of this file is passed as an
argument in the global 'argv' array, by the command interpreter.
     infile = fopen(argv[2],"r");
     if (infile == NULL){
          cout << form("sim: could not open %s",argv[2]);
          return;
     }

// Get the number of units and the number of patterns.
     fscanf(infile,"%d %d",&unitCount,&patCount);
// Create space for input, output and target units
     AllocateUnits(3*unitCount);
// Make the input units.
     for (i=0; i<unitCount; i++){
          MakeUnit("INPUT",0,0,0,0,0,NULL,0);
// No Unit activation function.
          SetFlag(i,NO_UNITFUNC_FLAG);
// No Site activation function.
          SetFlag(i,NO_SITEFUNC_FLAG);
// No Link activation function.
          SetFlag(i,NO_LINKFUNC_FLAG);
     }

// Name them as a vector - In.

     NameUnit("In",VECTOR,0,unitCount);
```

Figure 3.3 (contd.)

```
// Make a set of the input units.
     DeclareSet("InputSet");
     AddToSet("InputSet",0,unitCount-1);
// Make the output units.
     for (i=unitCount; i<2*unitCount; i++){
          MakeUnit("OUTPUT",0,0,0,0,0,UFpass,0);
// Add a site with name "feed".
          MakeSite(i,"feed",SFweightedSum,0);
          SetFlag(i,NO_LINKFUNC_FLAG);
     }
     NameUnit("Out",VECTOR,unitCount,unitCount);
     DeclareSet("OutputSet");
     AddToSet("OutputSet",unitCount,2*unitCount-1);
// Make the target units.
     for (i=2*unitCount; i<3*unitCount; i++){
          MakeUnit("TEACHER",0,0,0,0,0,NULL,0);
          SetFlag(i,NO_UNITFUNC_FLAG);
          SetFlag(i,NO_SITEFUNC_FLAG);
          SetFlag(i,NO_LINKFUNC_FLAG);
     }
     NameUnit("Teach",VECTOR,2*unitCount,unitCount);
     DeclareSet("TeachSet");
     AddToSet("TeachSet",2*unitCount,3*unitCount-1);

/* Make links from input units to site "feed" at output units with
weight = 0 */

     for (i=0; i<unitCount; i++)
          for (j=unitCount; j<2*unitCount; j++)
               MakeLink(i,j,"feed",0,0,NULL);
```

Figure 3.3 (contd.)

```
/* Make links from target units to site "teach" at output units,
weight = 100 */
     for (i=unitCount; i<2*unitCount; i++)
          MakeLink(i+unitCount,i,"teach",1000,0,NULL);
}
/* Function for setting the pattern for the input units. This C++
function may be easily rewritten for the C compiler by using
'malloc' in place of 'new' and 'printf' in place of 'cout <<' */

void ReadPat(int mode){
static int readInput = FALSE;
static int CurPat=0;
int i,j,n;
// To read the input file and create the arrays only
// on the first call to this function.
     if (!readInput){
          readInput = TRUE;
/**** This has to be done in place of a simple declaration such
as: int ipat[m][n] because the number of units and patterns will
be known only at run time. */
          ipat = new intp[patCount];
// Allocate array for input patterns...
          for (i=0; i<patCount; i++)
               ipat[i] = new int[unitCount];
// ...and the target patterns.
          tpat = new intp[patCount];
          for (i=0; i<patCount; i++)
               tpat[i] = new int[unitCount];
/****/
// Read in the input patterns...
          for (i=0; i<patCount; i++)
               for (j=0; j<unitCount; j++)
```

Figure 3.3 (contd.)

```
                    if (!(n = fscanf(infile,"%d ",&ipat[i][j]))){
                         cout << "Premature end of file.\n";
                         exit(1);
                    }
// ...and the target patterns.
          for (i=0; i<patCount; i++)
               for (j=0; j<unitCount; j++)
                    if (!(n = fscanf(infile,"%d ",&tpat[i][j]))){
                         cout << "Premature end of file.\n";
                         exit(1);
                    }
 // Display the read patterns.
          cout << "Input Pattern Set:\n";
          for (i=0; i<patCount; i++){
               for (j=0; j<unitCount; j++)
                    cout << form("%d ",ipat[i][j]);
               cout << "\n";
          }
          cout << "\n";
          cout << "Target Pattern Set:\n";
          for (i=0; i<patCount; i++){
               for (j=0; j<unitCount; j++)
                    cout << form("%d ",tpat[i][j]);
               cout << "\n";
          }
     }
// All of the above done only once.
/* On each call set input vector on input  units  and  0  on  each
output unit. */
     for (j=0; j<unitCount; j++){
          SetOutput(j,ipat[CurPat][j]);
          SetOutput(j+unitCount,0);
     }
```

Figure 3.3 (contd.)

```
// Set target vector on target units only if called during simulation.
     if (mode == SIM)
          for (j=0; j<unitCount; j++)
               SetOutput(j+2*unitCount,tpat[CurPat][j]);
     else
          for (j=0; j<unitCount; j++)
               SetOutput(j+2*unitCount,0);
     CurPat++;
// To cycle over the same set of patterns.
     CurPat %= patCount;
}

/* The Learning Rule.   */
void LearnPass(){
Site *sp;
Link *lp;
     for (int i=unitCount; i<2*unitCount; i++)
          if(GetOut(i) != GetOut(i+unitCount)){
               lp = GetLinks(i,"feed");
               for ( ; lp; lp=lp->next)
                    if (GetLinkValueP(lp))
                         if (GetOut(i+unitCount))
                              SetWeightP(lp,GetWeightP(lp) + ETA);
                         else
                              SetWeightP(lp,GetWeightP(lp) - ETA);
          }
}

/* The Unit Activation function for the output units. */
UFpass(Unit *up){
Site *sp;
sp = GetSitePtrP(up,"feed");
```

Figure 3.3 (contd.)

```
        if (GetSiteValueP(sp) > 0){
                SetPotP(up,1);
                SetOutP(up,1);
        }
        else{
                SetPotP(up,0);
                SetOutP(up,0);
        }
}

/* Function to print the final values for the weights, after simulation.
void PrintWts(){
Site *sp;
Link *lp;
        cout << "\n";
        cout << "The final values for the weights are:\n";
        for (int j=unitCount; j<2*unitCount; j++){
                sp = GetSitePtr(j,"feed");
                for (lp=GetLinksP(sp); lp; lp=lp->next)
                        cout << form("%5d ",GetWeightP(lp));
                cout << "\n";
        }
}
```

Fig 3.3

The function *ReadPat*, which is the function for reading the input and target vectors from a file, storing them into an input array and a target array and then clamping the appropriate values on the different units, has been so written that reading the patterns from the file and storing them into arrays is done only the first time the function is invoked. Subsequent calls will skip this action. Each time the function is called, the single argument tells the function whether it has been called during simulation or during testing. This will help the function decide whether or not

to clamp the target inputs. A few more points to note about the user-written functions will be mentioned shortly.

Once these functions have been written, they may be compiled with the simulator code by typing the *simgen* command with the name of the file containing these functions as an argument, at the system's shell prompt(remember that the simulator has not yet been started). This creates the simulator executable *sim*. The simulator is now started by running *sim*. The simulator start up message is displayed followed by the simulator prompt "->". The user can start typing the simulator commands. Network construction is first done by calling the *BuildNet* function with the name of the file containing the data (input and target patterns) as the argument:

call BuildNet data <RETURN>

Then the *network input function* and the *learn function* are specified:

setinputfunc ReadPat <RETURN>

setlearnfunc LearnPass <RETURN>

Now, the simulation is run:

simulate 30 5 <RETURN>

runs the simulation for 30 steps of 5 iterations. The network is then tested. The *list* and *pause* commands aid in examining the values set on the input units and the corresponding pattern developed on the output units:

list on <RETURN>

list + all <RETURN>

test 3 5 <RETURN>

Finally, the weights developed on the connections from the input units and the output units can be printed as a matrix by calling the *PrintWts* function:

call PrintWts <RETURN>

This is an outline of the process of constructing a network and simulating it. The examples of Chapter 4 together with the code for the examples will help in clarifying these aspects further. of course, a cursory examination of the Users' Manual should also prove to be very helpful in carrying out these and other simulations.

3.13 Other Salient Features:

The following paragraphs briefly outline important features of the simulator.

3.13.1 User-written Functions:

All functions that are specified by the user (input function, network building function, learning functions, the activation function and those written for other purposes) must necessarily be of type *void* i.e. they cannot return any values. Of these, the activation functions and the network input and learning functions are invoked from within the system and hence, must conform to some requirements that are explained below. The network construction function and other functions are invoked by the user from the command interface. Any arguments required by these commands are typed on the command line following the name of the command, and are passed to the function via a global *argc-argv*-like structure. The user function can obtain the number of arguments from the global variable *argc* and the arguments themselves from the *argv* array. It is entirely the responsibility of the user function to

check the number and type of arguments, and retrieve them appropriately. This has been done to provide maximum flexibility to the user in designing his functions.

The unit,site and link activation functions are always invoked by the system with a single argument viz. the pointer to the unit, site or link object that will be updated by the function. In view of this, the user functions should be written such that access to the unit, site or link objects is through this pointer and using appropriate functions. This is illustrated by the *UFpass* unit activation function of the preceding example. The learn function is not passed any arguments. All data used by it are assumed to be available globally.

Another user-written function that is invoked by the system, the network input function, is called with a single integer argument that functions as a flag to indicate to the input function, whether it has been called during simulation or during testing; generally, the input function is required to behave slightly differently in the two cases.

Important Note: All user-specified functions should have names that begin with upper-case letters. This restriction is imposed to limit the size of the system's name table. See 3.13.

3.13.2 The Command Interface:

When the simulator is started, it automatically enters the command interface and displays the simulator prompt. Then the user may then type in the simulator commands to carry out the desired operations. The command that is typed in is parsed and checked for syntactic validity; if the command is found syntactically correct, the corresponding action is carried out, else a terse error message is displayed. The user can find out the exact syntax of the command using the online help facility.

The command interface, and the simulator, can be exited by typing the *quit* command. Typing *Ctrl-C* when the execution of a

command is in progress will abort the command and return control to the command interpreter i.e. the simulator will now be ready to accept further commands. However *Ctrl-C* should be sparingly used while a command is in execution, since this could affect the consistency of the simulator data structures such as the linked lists representing the sites attached to a unit or links attached to a site, in the case of commands which manipulate these.

Arguments to Commands: The arguments to the commands are typed on the command line following the name of the command and separated by blanks. The names of units, unit types, sites, sets, functions etc. need (should) not been closed in quotes. The only arguments that need to be enclosed in quotes are the string arguments to the *pipe*, *print* and *printpause* commands apart from any string arguments required by an user function.Note: The command interpreter recognizes certain words as *keywords* used by the system. These words are *SCALAR*, *VECTOR*, *ARRAY*, *RANDOM*, *UNITS*, *SITES*,*LINKS*, *ALL*, *ON*, *STEP*, *POT*, *SET*, *ALL*, *ON* and *OFF* (in lower case as well) besides the names of the commands themselves. These cannot be used for names.

Unit Specification in Commands: When a single unit has to be specified as an argument to a command, either its index or its name can be used.

e.g.

displayunit 1 <RETURN>

displayunit Control <RETURN>

displayunit Auto[3][3] <RETURN>

are all acceptable.

It is perfectly valid to specify a collection of units in place of a single unit in any command. A group of units can be specified as a range of units (consecutive units in the range will be assumed) or as a set name, in which case all members of the set will be involved in the specified operation. In specifying the range of

units, either method for specifying single units (index or name) or a combination of these can be used. Thus,

displayunit 1 - 10 <RETURN>

displayunit Fire - 10 <RETURN>

displayunit Input[0] - Input[5] <RETURN>

displayunit Set_A <RETURN> (where *Set_A* is a set)

are all acceptable.
In addition, the keyword *all* is taken to mean all units that exist. (Keyword *all* used in place of a site name means all sites at the specified unit(s).)

Help Information: The entire list of commands available at the command interface, their abbreviations, syntax and other details can be obtained online with the *help* command. Typing in just *help* displays the list of commands along with abbreviations that may be used in their place. The syntax of the command and an explanation of the action of the command is obtained by typing *help* followed by the name of the command on the command line.

Command Logging and Command Files: The commands typed in by the user will be automatically logged by the system in a temporary file. At the end of the session, before exiting, the simulator asks the user if he wants the log file to be saved. If so, the user can specify the name of the file in which to save the logged commands. If no name is specified by the user, the simulator generates a unique file name and saves the logged commands in that file. Whenever the user needs to repeat the same operations, he can ask the system to read the commands from the previously saved log file. The system reads the commands in the file and automatically executes them. This file can be edited if a slightly modified set of actions need to be performed.Alternatively, the user himself can create the file of the commands that need to be

executed. Command files are executed with the *read* command:
read <command_filename> <RETURN>

3.13.3 Debugging Support:

Apart from an elaborate set of display commands that could be very helpful in debugging, the system supports a *break* command with which the simulation may be temporarily suspended and control handed back to the user to allow him to examine the network parameters, or even to modify some of the system or network parameters:

break <steps> <RETURN>
After the indicated number of steps have been run, the simulator exits to a debug interface and prints the debug prompt "# ". All the commands that are acceptable at the top-level command interface also work here. Simulation may be resumed with the *continue* command:

continue [<steps>] <RETURN>

The optional argument to the *continue* command tells the simulator to continue the simulation for that many more steps and then break again. If no argument is specified, the simulation runs to completion.

Another related, useful feature is that the user can specify a set of commands to be automatically executed when the simulator *breaks* to the debug interface. This saves the user the trouble of having to retype the same set of commands, each time control is returned to the debug interface.The commands can be specified with the *do* command:

First, simply type *do <RETURN>* against the prompt followed by each of the commands terminated with a *<RETURN>*. End the sequence of commands with *end*. The system will remember these

commands and execute them each time control returns to the debug interface. However, the execution of these commands may be terminated with the *delete* command.

3.13.4 Integer and Floating-Point Versions of the Simulator:

The values for unit potentials and outputs, the *value* fields of the *Site* and Link objects, the link weights, and the unit, site and link data fields are generally assumed to be integers. A scaling factor (say, 1000) may be used with the weights for the links to represent values in the range -1 to+1 (which is typically the case). This is quite feasible for the simulation of a majority of networks. Such an assumption results in a simulator which runs faster, especially on a computer without hardware floating-point support, and consumes less space, especially when large networks are involved.

However, if the user wishes to use floating-point values for the above-mentioned fields, a floating-point version of the simulator may be generated by specifying the "*-f*" option to the *simgen* command. This results in all the system and user code to be compiled with the *-DFSIM* option to the C++ compiler which generates code that treats these fields as floating point values.

The type *FLINT* is defined to be either an *int* or *float* at the time of compilation, depending upon whether the *FSIM* flag is defined or not. If the floating-point option is required, the user should bear in mind the above fact and use the type *FLINT* for the fields indicated above and use conditional compilation to generate correct code. An example of this can be found in the code for simulations given in Appendix B. In the preceding discussion and in the rest of the book, for the sake of simplicity, these fields are assumed to be of integer type, which is also the default type.(For more details about conditional compilation the user is referred to the C compiler manual page.)

3.13.5 Implementation Notes:

A couple of important aspects of the implementation have been mentioned below. The user is seldom concerned with these details. However, they may be helpful in modifications/enhancements to the system that may be contemplated in the future.

(i) *Outputs Array*: In addition to the unit array, as part of the network data structure, there is an one-dimensional array, Outputs, with the same number of elements as the unit array. It is used hold the output values of the units during simulation, and it is from this array, rather than directly from the output of the source unit, that a link obtains its input. (The value field of a link points to the element whose index in this array corresponds to that of the source unit.) The reason is that, if the simulation mode is synchronous, the output of an unit after updation should not be made available to the units to which it is connected, until the next time step. At the end of a step of simulation, the outputs of the units are copied into the Outputs array, so that they become available to the other units during the next step of simulation.

(ii) *Command Processing*: The command interpreter has been implemented using Lex and Yacc for parsing the commands. The action routines of Yacc are restricted to collecting the arguments into the global *argc-argv*-like structure (described previously) and invoking functions that do the actual processing of the commands. These functions collect the arguments from the *argv* array and use them for their processing. In most cases, these functions invoke lower-level functions to perform their actions. Part of the checking for the validity of the arguments is done in the course of parsing, with the remaining checks done by the lower-level functions. Even though this may seem slightly roundabout, it was preferred for the sake of the conceptual simplicity and the modularity that it provides. It makes it very easy to add new commands to the system by following

the same structure and without, in anyway, affecting the existing commands. Its effect on the efficiency of the system is, at best, marginal.

(iii) *Name Table*: A hash table is used by the system for maintaining names used within the system. These include names of units, unit types, sites,sets, states and functions used by the system. With unit names the system stores the type of the name (whether *SCALAR*, *VECTOR* or *ARRAY*) and other indexing information.

The pointers to the user-specified functions used in the course of network construction and simulation, is collected into the name table from the simulator executable file when the simulator starts up. This is done with the *UNIX* *nm* command. The name information provided by this command contains many other functions in addition to the user-specified ones, and no information to distinguish them. However, a large majority of these other functions have names that begin with a lower case letter. Hence, in order to filter out these unwanted names and, thereby limit the size of the name table, the restriction that all user-specified functions have to have names that begin with a upper case letter has been imposed. A large name table not only occupies more space, but also increases search times.

Chapter 4

SIMULATIONS

This chapter summarizes the result of several simulations which were performed using the ConnectSim++ system. The main aim of running these simulations has been to see how effectively ConnectSim++ can be used as a general purpose connectionist network simulation tool. As such, the choice of problems have been dictated by the following considerations:

* Connectionist networks with varied architectures have been chosen. Thus, completely connected, two-layered and multi-layered networks can be found in the examples. The examples also include networks which use different information processing techniques like settling and feedforward processing.

* Examples have been chosen such that each one uses an entirely different learning algorithm.

In all six simulations have been tried. The following sections describe each problem, along with simulation results. The actual code required to define these networks and run the simulations is listed in Appendix B.

4.1 Pattern Associator:

The Network. A *pattern associator* is a network in which a pattern of activity over one set of units (*input units*) causes a pattern of activity over another set of units (*output units*). Thus, the pattern associator is basically a two-layered network which maps a set of input patterns to a set of output patterns.

The model we use consists of eight input and eight output units and a set of connections from each input to each output

unit. The network is illustrated in figure 4.1, where active units have been darkened.

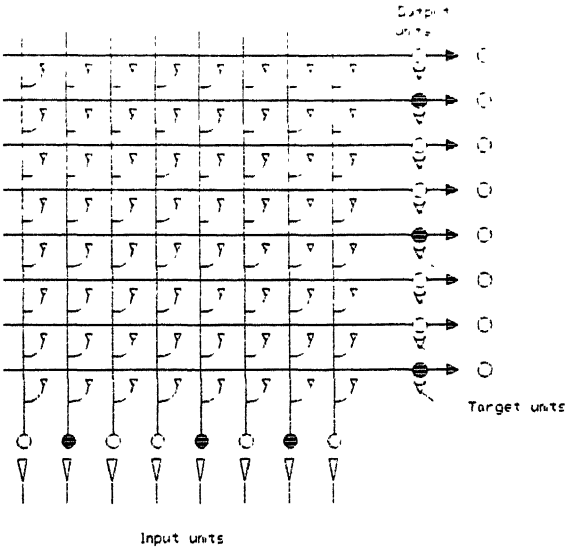


Figure 4.1
Pattern Associator

Input-output patterns. All units in the network are binary - they are either active (1) or inactive (Ø).

The pattern associator network is required to learn the rule of 78 [Rumelhart & McClelland, 1986]. This rule is as follows:

* Input patterns consist of one active unit from each of the following sets:

  (1 2 3)
  (4 5 6)
  (7 8)

* The output pattern paired with a given input pattern consists of:

  the same unit from (1 2 3)
  the same unit from (4 5 6)
  the other unit from (7 8)

Example: 3 6 7 ⟶ 3 6 8
The input pattern has units 3, 6 and 7 active. The associated output pattern has units 3, 6 and 8 active.

Updating rules. Each output unit calculates the net input to it using all of the weighted connections from the input units. The

net input is simply the sum, over all input units, of the output value of the input unit times the corresponding weight. Algebraically, the net input to output unit j is:

$$net_j = \sum_i w_{ji} o_i$$

where o_i is the output of input unit i and w_{ji} represents the weight from unit i to unit j. The output of unit j is fixed by:

$$o_j = \begin{cases} 1 & net_j > \emptyset \\ \emptyset & net_j \leq \emptyset \end{cases}$$

The input pattern is clamped on the input units and the desired output pattern is clamped on the target units.

Learning rule. Connection strengths are adjusted using the *perceptron convergence procedure* (Section 2.4 (1)). On a learning trial, the network is presented with both the input pattern and the target output pattern. As on a test trial, the pattern associator network computes the output it would generate from the input. Then, for each output unit, the model compares its answer with the target. The target supplies a sort of teaching input to each output unit, telling it what it ought to have. Weights are modified as follows:

* If the actual output matches the target output, no changes are made, since the network is doing the right thing.

* When the computed output is Ø and the target says it should be 1, we want to increase the net inplut so that the unit will be active the next time the same input pattern is presented. To do this, we increase the weights from all of the input units that are active by a small amount η.

* When the computed output is 1 and the target specifies Ø, we want to decrease the net input. To do this,the weights from all of the input units that are active are reduced by η.

Simulation results. The simulation was run under the following conditions:

* All weights were initialised to zero.

* During the learning period, all the 18 possible patterns specified by the rule of 78 were presented to the network. This procedure was repeated twice.

The resulting weight matrix, after two cycles of exposure to all the 18 patterns, is shown below:

		input units							
		u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8
output units	u_1	1	-1	Ø	Ø	Ø	Ø	Ø	Ø
	u_2	Ø	1	-1	Ø	Ø	Ø	Ø	Ø
	u_3	-1	Ø	1	Ø	Ø	Ø	Ø	Ø
	u_4	Ø	Ø	Ø	1	-1	Ø	Ø	Ø
	u_5	Ø	Ø	Ø	Ø	1	-1	Ø	Ø
	u_6	1	Ø	-1	-1	-1	2	Ø	Ø
	u_7	Ø	Ø	Ø	Ø	Ø	Ø	-2	2
	u_8	Ø	Ø	Ø	Ø	Ø	Ø	1	-1

Input units are indexed by column and output units are indexed by row. Thus, the entry in the i^{th} row of the j^{th} column, w_{ij}, indicates the weight of the connection from input unit u_j to output unit u_i.

Note that the resulting weight matrix is quite close to the optimal set of weights required to capture the rule of 78.

4.2 Auto Associator:

The Network. An auto associator [Hinton, 1987] is a kind of pattern associator in which the input units are identical with the output units. Such a network can associate an input pattern with itself. Whenever a portion of the input pattern is presented, the

auto associator can retrieve the entire pattern.

The auto associator is a completely connected network with self-loops (figure 4.2). As such, an iterative process is required for the network to settle down into a stable state - the network uses *settling* (section 2.3.9).

The network chosen for this example has eight units.

Input patterns. Since the input and output units are identical in an auto associator, the input and output patterns are also identical. The input pattern, therefore, plays both the role of the teaching input and of the pattern to be associated.

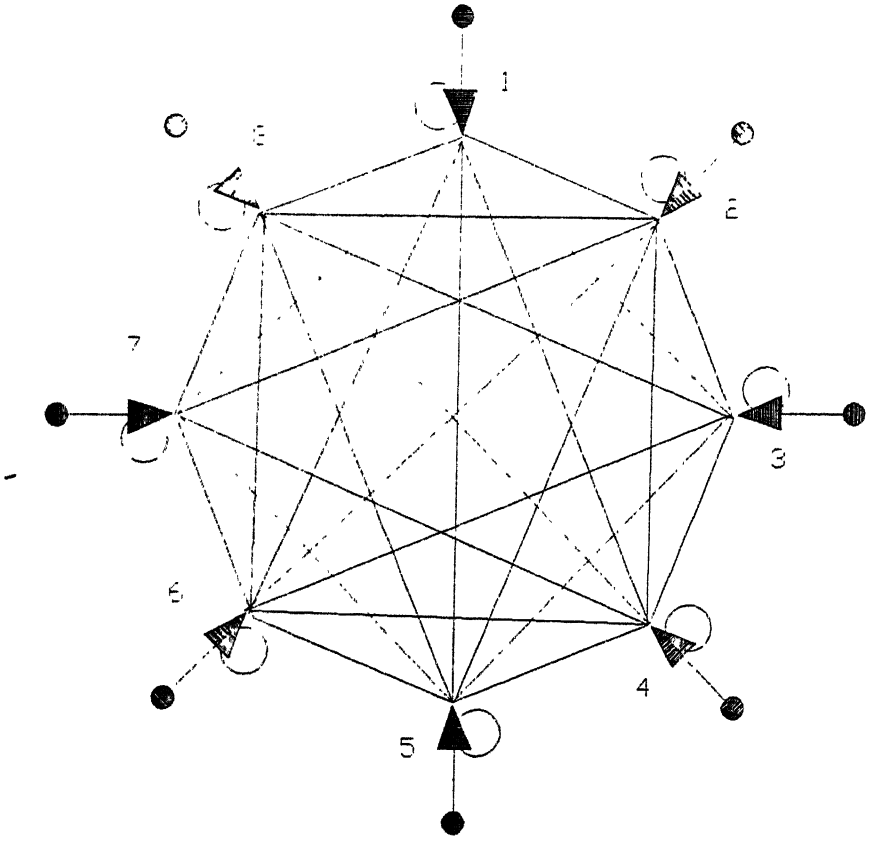


Figure 4.2 Auto Associator

All units in the system can have one of three permitted output values: -1, Ø and +1. -1 signifies the "off" state while +1 signifies the "on" state. When the state of a unit is to be left unspecified - it is neither "on" nor "off", a unit whose output is "floating" - its output value is set to Ø. An incompletely

specified input pattern contains Ø's.

We have used the auto associator to store two patterns[1] whose components are all -1 or +1:

	input/output pattern							
1.	-1	1	-1	1	-1	1	-1	1
2.	1	1	-1	-1	-1	-1	1	1

Updating rules. The updating rule used for the auto associator units is similar to the one used for the pattern associator except for the introduction of a threshold. We would, therefore, have for unit u_j:

$$net_j = \sum_i w_{ji} o_i - \theta_j$$

where net_j is the total input for u_j, o_i is the output for unit i, w_{ji} is the connection strength from u_i to u_j and θ_j is the threshold for unit j. As usual, the output of unit j is

$$o_j = \begin{cases} 1 & net_j > Ø \\ Ø & net_j \leq Ø \end{cases}$$

Learning rule. The network uses the *Hebbian learning rule* (Section 2.4 (1)) for modifying connection weights. According to this rule, weight w_{ji} of the connection from u_i to u_j is changed by an amount Δw_{ji} proportional to the product of the activation a_j of u_j and the output o_i of u_i:

$$\Delta w_{ji} = \eta a_j o_i$$

---

[1] An auto associator such as this has very limited capacity. A network with eight units cannot reliably store more than two patterns. Moreover, the patterns should be linearly independent. See the subsection on the theory of auto associators.

η is the proportionality constant. Here, the activation of a unit and its output are identical. Hence, Hebb's rule reduces to

$$\Delta w_{ji} = \eta\, o_j\, o_i$$

with o_j and o_i representing the outputs of units j and i respectively.

Brief theory. An auto associator network of the form described here, which uses Hebbian learning, is termed a *Hopfield net.* Hopfield nets store vectors whose components are all +1 or -1. To retrieve a stored vector from a partial description (which is a vector containing some zero components), the network is started at the state specifed by the partial description and then repeatedly updated, one unit at a time. The units can be chosen in random order or in any other order provided each unit is chosen finitely often. Hopfield has shown that the behaviour of the network is governed by a *global energy function*

$$E = -\frac{1}{2} \sum_{i,j} o_i\, o_j\, w_{ij} + \sum_i o_i\, \theta_i$$

where the symbols have their usual meanings. It can be shown that each time a unit updates its state, it adopts the state that minimizes this energy function. As units are updated, the energy must decrease until the network settles into a local minimum of the energy function. Thus, the retrieval process can be viewed as follows: The weights define an "energy landscape" over global states of the network and the stored vectors are local minima in this landscape. The retrieval process consists of moving downhill from a starting point to a nearby local minimum.

If too many vectors are stored, there may be spurious local minima caused by interactions between the stored vectors. Also, the basins of attraction around the correct minima may be long and narrow instead of round, so a downhill path from a random starting point may not lead to the nearest local minimum. For random n-component vectors, the storage capacity of Hopfield net is only 1/klog(n) bits per weight.

Simulation results. The simulation was started with all weights, except those for the input lines, set to zero. The weights for the input lines are initially set to a very high value so that the input presented always comes as the output (except for unspecified units). The learning procedure trains the other weights in such a way that the given input patterns can be retrieved even if they are partially specified. As mentioned elsewhere, the threshold of a unit is implemented as the negative of the weight from a unit which is always "on" (+1). Thus threshold values are also learned, just like ordinary weights.

Hebb's rule being a "one-shot" learning procedure, the network is shown the two input patterns just once. Since updating the network involves a settling process, the simulation uses iterative updating. Actually, the *'simulate with itererr'* command was used with iteration error of zero. The resulting weights and thresholds, after learning, are listed below.

	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8	θ	w_{in}
u_1	2	Ø	Ø	-2	Ø	-2	2	Ø	Ø	5Ø
u_2	Ø	2	-2	Ø	-2	Ø	Ø	2	-2	5Ø
u_3	Ø	-2	2	Ø	2	Ø	Ø	-2	2	5Ø
u_4	-2	Ø	Ø	2	Ø	2	-2	Ø	Ø	5Ø
u_5	Ø	-2	2	Ø	2	Ø	Ø	-2	2	5Ø
u_6	-2	Ø	Ø	2	Ø	2	-2	Ø	Ø	5Ø
u_7	2	Ø	Ø	-2	Ø	-2	2	Ø	Ø	5Ø
u_8	Ø	2	-2	Ø	-2	Ø	Ø	2	-2	5Ø

The first matrix indicates the weights on the connections between units, with w_{ij} giving the weight of the connection from u_j to u_i. Note that the units indexed by rows and columns refer to the same unit[2] . The second matrix gives the threshold θ and weight w_{in} of the input connection from each unit.

---

[2]This is unlike the situation in the pattern associator case(section 4.1),where input units are indexed by columns and output units by rows.

The performance of the network was tested (using 'test with itererr') by presenting some incompletely specified input patterns:

	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8
input	-1	1	Ø	Ø	Ø	Ø	Ø	Ø
output	-1	1	-1	1	-1	1	-1	1
input	Ø	Ø	Ø	-1	-1	Ø	Ø	Ø
output	1	1	-1	-1	-1	-1	1	1
input	Ø	Ø	Ø	-1	Ø	Ø	Ø	Ø
output	1	1	-1	-1	-1	-1	1	1
input	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø
output	-1	1	-1	1	-1	1	-1	1
input	Ø	1	Ø	Ø	Ø	Ø	Ø	1
output	-1	1	-1	1	-1	1	-1	1
input	Ø	Ø	-1	Ø	-1	Ø	Ø	Ø
output	-1	1	-1	1	-1	1	-1	1

In each case, the outputs of all units have been set to zero before giving the partially specified input pattern. This is to provide a uniform starting point in the energy landscape. Moreover, the energy surface may be such that it may not be possible to reach the required local minimum if we start at some random point.

4.3 The Dipole Experiment:

The Network. The network is constituted by a 4 x 4 array of input units, each of which is connected to an inhibitory cluster of two units(figure 4.3). This network is used to illustrate how the competitive learning mechanism (See Section 2.2 (2)) can discover structure in the stimulus patterns presented [Rumelhart & Zipser, 1986].

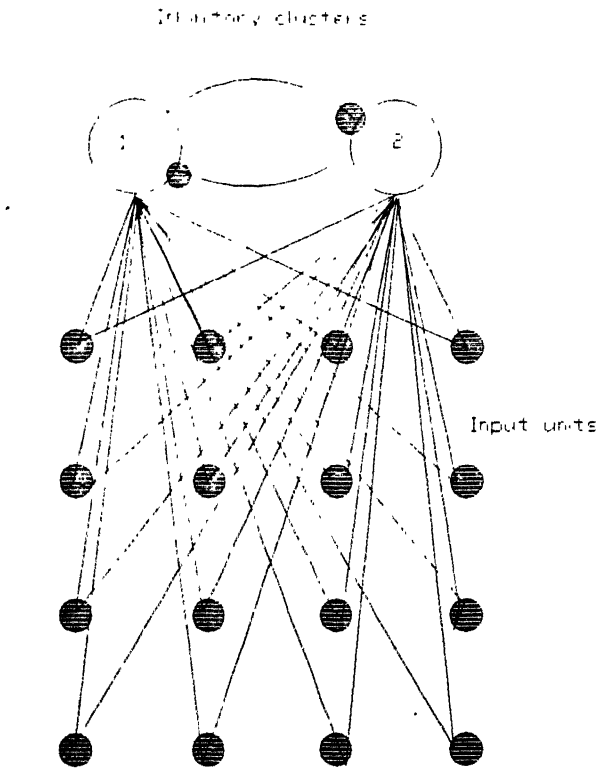


Figure 4.3 Competitive Learning.

Input patterns. The input patterns to this network consist of dipoles - stimulus patterns having exactly two active elements with all other elements inactive. Further, not all possible dipoles are acceptable input patterns. The stimulus space is limited to dipoles in which the active units form adjacent pairs in the grid. In all, there are 24 possible adjacent dipole patterns defined on the 4 x 4 input grid.

The competitive learning procedure is a type of unsupervised learning method. The network, therefore, has no "preferred output" or "correct output". We just present the input patterns and see if the learning algorithm can discover any structure in these stimulus patterns.

An active unit has an output of 1; output values of Ø represent inactive units.

Learning algorithm and updating rule. These are exactly as specified in section 2.4 (2) with the proportionality constant $g = \emptyset.1$ and the number of active units in pattern S_k

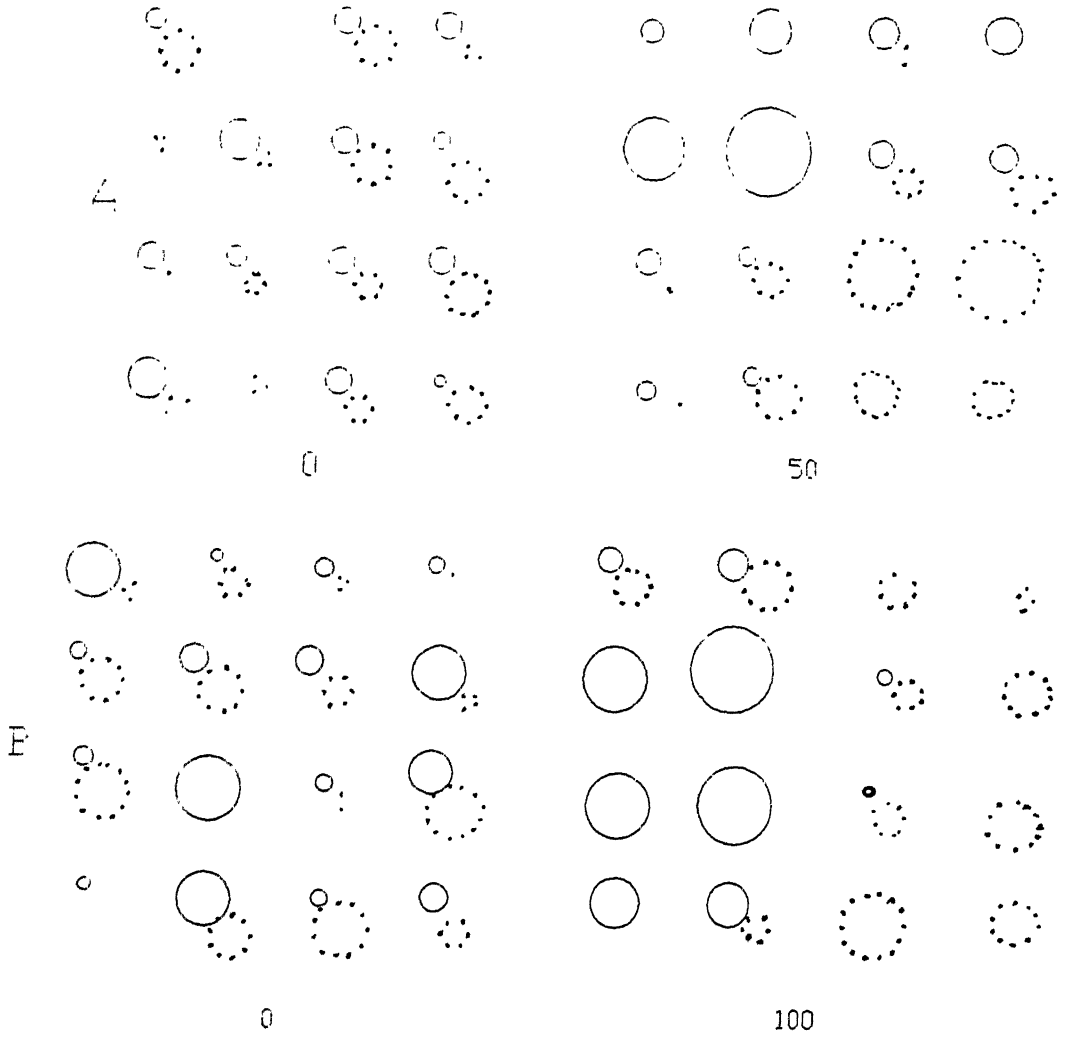
$$n_k = \sum_i c_{ik} = 2.$$

Simulation results. As mentioned in section 2.4 (2), the weights for unit u_j in the inhibitory cluster were initialised such that $\sum_i w_{ji} = 1$. Here i ranges over all the input units. The input consisted of repeatedly presenting one of the 24 adjacent dipole patterns at *random.*

The results of two runs are indicated in figures 4.4 and 4.5. The values are shown laid on a 4 x 4 grid so that weights are next to one another if the units with which they connect are next to one another.

The results clearly show that the weights move from a rather chaotic initial arrangement to a much more orderly distribution in which each unit has chosen a coherent half (approximately) of the grid to which they respond. As far as the competitive learning mechanism is concerned, the input lines are unordered. The two-dimensional grid-like arrangement exists only in the statistics of the population of stimulus patterns. Thus, the system has *discovered* the dimensional structure inherent in the stimulus population and has devised binary feature detectors to tell which half of the grid contains the stimulus pattern.

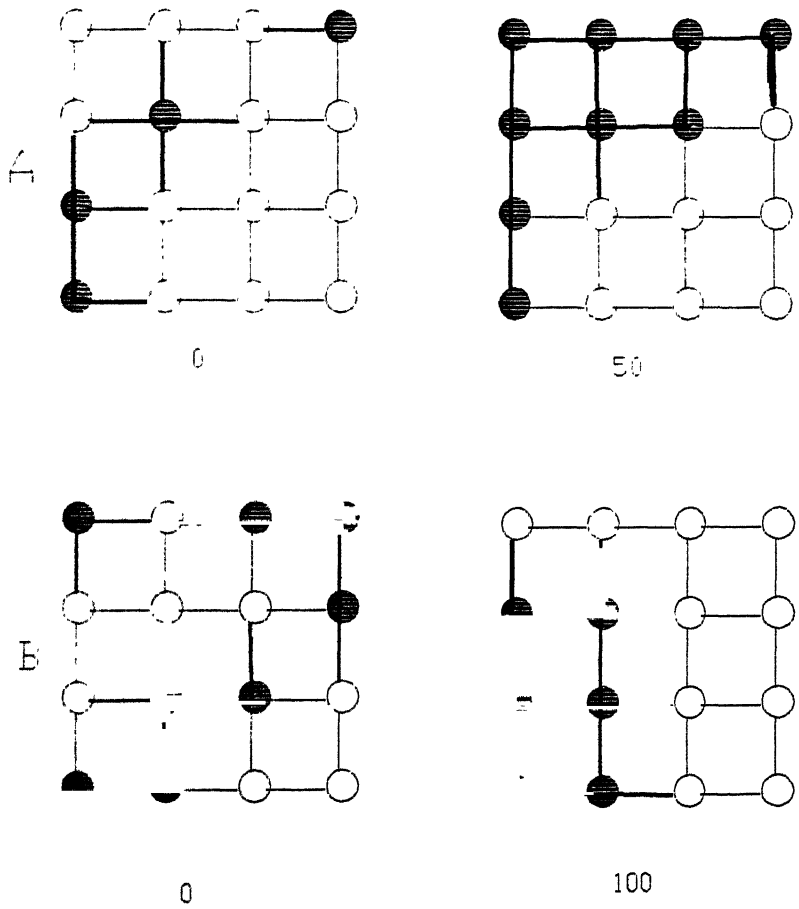
Finally, note that the weight patterns pictured have not yet stabilised,in the sense that running the simualtion for more trials can change the distribution of weights. Stabilisation of the relative weights requires several hundred trials. Nevertheless, the self-organisation of the relative weights mentioned above is very clearly evident even at this early stage.



A : The results for one run with the dipole stimulus defined over a two-dimensional grid. The left-hand array shows the initial value of the weights and the right-hand array shows the weights after 50 trials. At any grid point, a complete circle indicates the weight from the corresponding input unit to unit 1 in the inhibitory cluster, while a dotted circle indicates the weight for unit 2. The radius of the circles is proportional to the weight.

B : Results for a second run, with 100 trials.

Figure 4.4 Results of competitive learning : Weights.



A : The results for one run with the dipole stimulus defined over a two-dimensional grid. The left-hand grid shows the relative values of the weights initially and the right-hand grid shows the relative values of the weights after 50 trials. A filled circle means that unit 1 had the larger weight on the corresponding input. An unfilled circle means that unit 2 hadthe larger weight. A heavy line connecting two circles means that unit 1 responded to the stimulus pattern consisting of the activation of the two circles, and a light line means that unit 2 won on the corresponding pattern. Note that two unfilled circles must always be joined by a narrow line and two filled circles must always be joined by a wide line.

B : Results for a second run, with 100 trials.

Figure 4.5 Results of competitive learning : Relative Weights.

4.4 Symmetry Detector

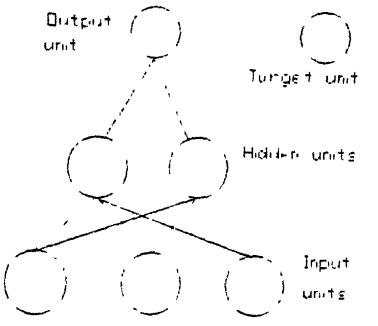


Figure 4.6 Symmetry Detector Network.

The Network. This three-layered network is used for classifying binary input strings. It determines whether or not they are symmetric about their center [Rumelhart, Hinton & Williams, 1986]. Since the network has three input units (figure 4.6), only three bit strings can be handled. Apart from the input units and single output unit, the network also has two hidden units. The network therefore has a multi-layered architecture. The environment specifies the input pattern and the expected output for a given input pattern. The task of discovering an appropriate *internal representation* for the hidden units is left to the learning algorithm.

Input-output patterns. The three-bit input patterns are clamped on the input units. Since the input strings are composed of Ø's and 1's, the states of input units are also restricted to Ø or 1. A similar restriction holds for the target unit.

The hidden units and output unit can have output values which range between Ø and 1. Moreover, the activation function for these units (Section 2.2 (3)) is such that the output value cannot reach its extreme values of 1 or Ø without infinitely large weights. Therefore, in a practical learning situation in which the desired outputs are binary (Ø,1) the system can never actually achieve these values. Therefore, we use the values of Ø.15 and

Ø.85 as the targets[a], even though we shall talk as if values of (Ø,1) are sought.

The input patterns and the corresponding target values are listed below:

input pattern			target value
u1	u2	u3	
Ø	Ø	Ø	1
Ø	Ø	1	Ø
Ø	1	Ø	1
Ø	1	1	Ø
1	Ø	Ø	Ø
1	Ø	1	1
1	1	Ø	Ø
1	1	1	1

On successful completion of learning, a symmetric input pattern should result in a output of 1 from the output unit.

Learning and activation rule. The symmetry detector network uses the generalised delta rule of section 2.2 (3), along with the logistic activation function. The simulation was run with a learning rate of η = Ø.5

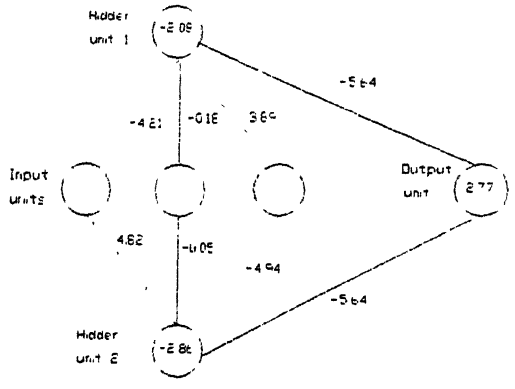
Simulation results. The system is started with the weights (and biases) initialised at random[4]. The weights chosen were real

---

[a] That is, any ouput $\leq$ 0.15 is considered a 0 and an ouput $\geq$ 0.85 is a 1.

[4] If all weights start out with equal values and if the solution requires that unequal weights be developed, the system can never learn. This is a problem with all networks using the generalised delta rule for learning.

numbers between +1 and -1. The learning being very slow, the whole set of 8 input patterns were presented several hundred times. Figure 4.7 shows the weights acquired after 500 presentations of each of the 8 three-bit patterns.



This network is the same as figure 4.6 but is organized differently for the sake of clarity. Bias values for the units are indicated within the circles representing the units.

Figure 4.7 Symmetry detector network after learning

Note the symmetry in the weights developed by the network. After learning, each of the 8 input patterns generated the following outputs:

input pattern			output value
u1	u2	u3	
0	0	0	0.86
0	0	1	0.11
0	1	0	0.88
0	1	1	0.13
1	0	0	0.10
1	0	1	0.88
1	1	0	0.10
1	1	1	0.89

4.5 XOR Network

The Network. The XOR problem is considered to be the "classic" problem for connectionist networks because it is one of the simplest which requires hidden units and because many other problems include XOR as a subproblem. The network we consider (figure 4.8) has two input units, one hidden unit and one output unit. Apart from normal connections, the model also has meta-connections (shown dotted in figure 4.8). The network has 100% meta-connectivity. Such a network has a meta-connection from

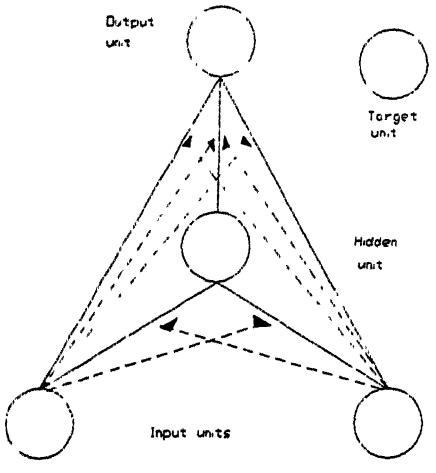


Figure 4.8 XOR network with meta-connections

each input unit to every non-meta connection in the network except those originating from the same input unit [Pomerleau, 1987].

Input-output pattern. The input consists of all possible two-bit patterns. The target output is the exclusive-or of the input pattern:

input pattern		desired output
0	0	0
0	1	1
1	0	1
1	1	0

The output unit and hidden units have output values which range between Ø and 1. The nature of the activation rule prevents these units from having outputs of exactly Ø or 1, unless the weights are infinitely large. We, therefore, accept anything below Ø.1 as Ø and anything above Ø.9 as 1, even though we continue to speak of these values as Ø and 1. In actual practice, a value of Ø.1 is used for Ø and Ø.9 is used for 1 in the input and target patterns also[5].

Learning rule and activation function. The meta-generalised delta rule of section 2.4 (4) is used. The activation rule applied is also indicated in that section.

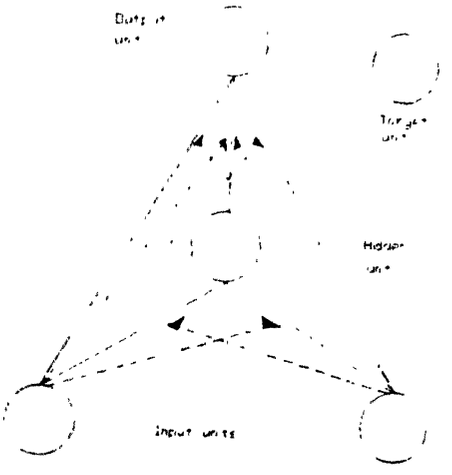
Simulation results. The meta-connections were implemented using the ConnectSim++ feature providing more than one connection to a site. A connection and all meta-connections to it were connected to the same site. A meta-connection is distinguished from an ordinary connection by storing the flag META (=-999) in the data field of the link representing the meta-connection. This is done at the time of network construction.

As for the symmetry detector network, weights of connections, bias values and weights of meta-connections are initialised to random values between -1 and +1. The learning here is faster than the case of the generalised delta rule. Figure 4.9 shows the weights acquired by the network after 3ØØ presentations of each of the 4 input patterns. Values within circles are the biases for the units.

The output values obtained for the 4 input patterns indicate that even though the network has learned the XOR rule, some more training is necessary (so that the first and last patterns result

---

[5] Failure to do so can result in a divide-by-zero error during simulation.



Bias values for the units are indicated within the circles representing the units.

Figure 4.9. XOR network after learning.

in an output less than or equal to Ø.1 and the other two patterns generate an output greeater than or equal to Ø.9).

input pattern		output
Ø	Ø	Ø.31
Ø	1	Ø.82
1	Ø	Ø.8Ø
1	1	Ø.11

4.6 The Four Coloring Problem:

The problem is essentially that of coloring a planarmap using no more than four colors, so that no neighboring regions have the same color . A solution does exist for the problem and the connectionist network that will be built is required to find solution [Goddard et al.]. The colors are RED, BLUE, GREEN and WHITE. Each region is represented with four units, one for each color. Since each region should should have exactly one color, only one of these units can be on at any one time. If some region

is shut out by neighbors' colors it may turn on some color anyway, which may force off some other region's color node. The general idea is for each region's units to inhibit each other (corresponding to the notion that a region can have at most one color), and for neighboring regions to inhibit each other from having the same color. Asynchronous simulation is used to ensure that the search space is explored until a stable state corresponding to correct coloring is found. The problem is, essentially, one that involves settling.

The units within a region will have inhibitory links to all others in the same region, with the weights highly negative (high inhibition). Also, no two neighboring regions can have the same color; hence, corresponding units (i.e., those representing the same color) of two regions having a border will have inhibitory links in either direction. For instance, if region X borders region Y, region X's blue unit will have an inhibitory link to region Y's blue unit, and vice versa. The same applies for the red, green and white units. Since the network should be allowed to search the space of possible colorings, neighboring regions should be able to have the same color for a short period of time, so the weight on these links will be moderate and negative (moderate inhibition).

The unit activation function will look at the inhibition arriving. If there is none, i.e., the region is not yet colored and no neighboring region is colored with the same color, then the unit will be turned on. If there is inhibition from the neighboring regions (mild inhibition) but not from unit within the same region, then with some small probability dependent on the strength of the inhibition, the unit will be turned on. If another unit in the region is on, the unit will remain off.

The map is specified using a data file containing the number of regions in the map and pairs of regions that have a common border with each other. Each region is represented by a unique number. The format of the data file is:

<number of regions>

<region number> <region number>

<region number> <region number>

<region number> <region number>

Results of the simulation. The network converged to a solution in about 20 steps for a map with five regions, with twelve pairs of regions having common borders.

Chapter 5

CONCLUSIONS

The capabilities of ConnectSim++, and the facilities offered by it for the definition and simulation of virtually any kind of network were described in sufficient detail in Chapter 3, and were illustrated with examples of a diverse nature showing the ability of the system to handle a broad spectrum of connectionist models. We conclude with a critical assessment of the achievements and what remains to be done. Those aspects of the system that could not be realized, mainly due to the limitation of time, and those that need to be strengthened to make the system more versatile and powerful, are highlighted.

5.1 ConnectSim++ - Characteristics:

The philosophy and objectives of ConnectSim++ has been outlined elsewhere in the thesis. In implementing these, it has drawn inspiration from two other connectionist network simulator systems with similar objectives: the SNAIL connectionist simulation system and the Rochester Connectionist Simulator. Despite the similarity of purpose, the two systems differ considerably in their approach to the problem.

ConnectSim++ has attempted to imbibe the merits of the two approaches. It has adopted the object-oriented paradigm of SNAIL that made it a highly flexible system and very easily extendable. At the same it has strived to gain the efficiency and speed advantages of the Rochester system. While the latter does provide scope for extension, it is rather restrictive as compared with the plasticity of SNAIL. The implementation of ConnectSim++ also embodies novel, and conceptually simple, ways to handle some tasks, the facility for debugging being an example.

In an overall assessment, it can be said that the system has, by and large, succeeded in achieving the stated objectives. It has

successfully provided a system that is flexible and user-friendly, in an object-oriented framework which encourages enhancements and modifications. We hasten to add that the system has been operational only for a very short time and is yet to reach serious users, whose feedback will eventually determine how successful the system has really been.

The following is a summary of the experience gained in the implementation and a very short period of using the system (mostly restricted to running the sample simulations):

(i) The framework provided by the system has been sufficiently general to build and test models of a radically differing nature - in terms of their structure, in terms of the problems that they are intended to solve and in the learning rules they employ. This is amply demonstrated by the simulation examples of Chapter 4 which have been deliberately chosen to cover models with widely varying and distinctive characteristics (for instance, the XOR network with the unconventional meta-connections). Almost any conceivable variation of these and other models proposed in the literature are implementable - adding new units in the middle of a simulation, links with propagation delays, weight changes or output changes that depend on the past values of the network parameters, weight decay, programmable weights etc.

(ii) Any drastic changes to the structure of one or more of the system objects - units, sites and links - that may be contemplated in the future can be easily achieved. This is due to the *inheritance* property of the C++ class which is used to represent these objects. A *derived class* of the class representing the object can be defined, which incorporates the required changes/enhancements, apart from inheriting the data objects and the functions of the base class. Further, the clean interface provided to the outside

by the C++ class construct simplifies the task of adapting the rest of the system to the newly incorporated changes.

(iii) The speed of the system is many orders of magnitude better than that of SNAIL, though no quantitative measurements have been made. Features are provided in the system to prune the operations involved in the simulation, eliminating any loss of efficiency. However, we suspect that there still exists some scope to squeeze out extra performance from the system.

(iv) The attempt to make the system flexible and general has made it necessary for the user to write some code for constructing the network, for providing inputs to the network and for specifying the activation functions and the learning rule. But these functions are often simple and straightforward. Further, these are specified using the widely used C (or C++) language.

(v) Running and controlling the simulation has been made simple with a set of very useful commands. The system also provides support for debugging the network.

5.2 ConnectSim++ - Shortcomings, Suggested Improvements and Extensions:

(i) The system presently provides a small library of activation functions and learning rules for ready use during simulation. But this library is, at best, skeletal. The power and usefulness of a system is greatly enhanced if a rich collection of such functions is supported. It eliminates the need for the user to write his own functions for most purposes; they can be simply called from the library. Sets of functions may be added to the library which make it possible to implement and test networks with advanced capabilities such as links with propagation

delays. Also, library functions that are tailored to specific architectures enable the user to construct networks based on these architectures simply by calls to appropriate library functions, thereby greatly reducing the programming effort.

(ii) A facility to checkpoint a simulation that is in progress and continue it later can prove very useful in simulating large networks whose simulation can take many hours. A related feature is the ability to save a constructed network into a file and load it at a later time for the purpose of simulation.

(iii) A graphic interface to the system with menu-driven facilities vastly improves the user-friendliness of the system. It would also simplify observing the dynamic behavior of the network and the process of debugging. If the graphic interface also supports multiple windows, it results in further convenience to the user, making it possible to type commands at a control panel while observing the output on a separate display panel.

(iv) The system may be extended to support simultaneous construction and testing of multiple networks with a provision to switch attention from one network to another. After testing the individual networks separately, the user should be able to set up connections between these networks, thereby making them subnetworks of a larger network. This way it is possible for the user to adopt a bottom-up approach to the design of complicated networks. This leads to the concept of *Super Units* which are actually parts of a network, but can be characterized as a unit.

REFERENCES

Feldman J.A. and *Ballard D.H.*, 1982, " Connectionist Models and their Properties ",Cognitive Science, Volume 6.

Grossberg S., 1978, " A Theory of Visual Coding, Memory and Development " In Leeuwenberg, E.L.J., and Buffart, H.F.J.M. (Eds). Formal Theories of Visual Perception New York, Wiley.

Goddard N.H., *Lynne K.J.* and *Mintz T*, 1989, " Rochester Connectionist Simulator ", University of Rochester Technical Report 233.

Hebb D.O., 1949, The Organization of Behaviour, New York, Wiley.*Hinton G.E.*, 1987, " Connectionist Learning Procedures", Carnegie Mellon University Technical Report CMU-CS-87-115.

Hinton G.E. and *Anderson J.A.*, 1981, " Parallel Models of Associative Memory ", NJ, Erlbaum, Hillsdale.

Hinton G.E., *Sejnowski T.J.* and *Ackley D.H.*, 1984, " Boltzmann Machines:Constraint Satisfaction Networks that Learn ", Carnegie Mellon University Technical Report CMU-CS-84-119.

Hopfield J.J., 1982, " Neural Networks and Physical Systems with Emergent Collective Computational Abilities ", Proceedings of National Academy of Sciences USA. Volume 79.

Kohonen T., 1984, Self-organization and Associative Memory, Berlin, Springer Verlag.

Lippmann R.P., 1987, " An Introduction to Computing with Neural Nets ", IEEE ASSP Magazine. April 1987.

Mani D.R., 1989, " SNAIL: A General Purpose Connectionist Network Simulator ",Dept. of Computer Science & Engineering, Indian Institute of Technology Kanpur.

Matheus C.J. and *Hohensee W.E.*, 1987, " Learning in Artificial Neural Systems ", University of Illinois at Urbana-Champaign Technical Report UIUC-R-87-1394.

McCulloh W.S. and *Pitts W.A.*, 1943, " A Logical Calculus of the Ideas Immanent in Nervous Activity ", Bulletin of Mathematical Biophysics, 5, 115-133.

Minsky M. and *Papert S.*, 1969, Perceptrons, Cambridge MA, MIT Press.

Pomerleau D.A.,1987, " The Meta-Genaralized Delta Rule: A New Algorithm forLearning in Connectionist Networks ", Carnegie Mellon University Technical Report CMU-CS-87-165

Rosenblatt F., 1958, " The Perceptron : A Probabilistic Model for Information Storage and Organization in the Brain ", Psychoanalytic Review, 65, 386-4Ø8.

Rosenblatt F., 1962, Principles of Neurodynamics, New York, Spartan.

Rumelhart D.E., *Hinton G.E.* and *McClelland J.L.*,1986, " A General Framework for Parallel Distributed Processing ", In Rumelhart D.E. and McClelland J.L.(Eds). Parallel Distributed Processing: Explorations in the Microstructures of Cognition, Volume 1. Cambridge, MA, MIT Press.

Rumelhart D.E., *Hinton G.E.* and *Williams R.J.*, 1986, " Learning Internal Representations by Error Propagation ", In Rumelhart D.E. and McClelland J.L.(Eds). Parallel Distributed Processing: Explorations in the Microstructures of Cognition, Volume 1. Cambridge, MA, MIT Press.

Rumelhart D.E., and *McClelland J.L.*, 1986, " On Learning the Past Tense of English Verbs ", In Rumelhart D.E. and McClelland J.L.(Eds). Parallel Distributed Processing: Explorations in the Microstructures of Cognition, Volume 2. Cambridge, MA, MIT Press.

Rumelhart D.E. and *Zipser D.*, 1986, " Feature Discovery by Competitive Learning ", In Rumelhart D.E. and McClelland J.L.(Eds). Parallel Distributed Processing: Explorations in the Microstructures of Cognition, Volume 1. Cambridge, MA, MIT Press.

Widrow G. and *Hoff M.E.*, 1960, " Adaptive Switching Circuits ", Institute of Radio Engineers, Western Electronic Show and Convention, Convention Record,Part 4, 96-104.

Appendix A

ConnectSim++ Users' Manual

ConnectSim++ is a general-purpose connectionist network (artificial neural network) simulator. It uses an object-oriented approach to implement the system objects (units, sites and links). The language used for the implementation is C++. The system has benefited in several ways from the use of the object-oriented approach (refer Chapters 1 and 3).

This write-up introduces the user to the organization of the system and the facilities provided by it to construct and simulate connectionist models of a wide variety. All the functions and commands provided by the system for this purpose, and for the purpose of observing the static and dynamic aspects of a network are explained. Also, the important global system variables that control the operation of the simulator, and certain other points about the implementation have been explained.

A preliminary knowledge of connectionist networks (their structure, operation, rules of learning etc.) and a knowledge of programming with C or C++ is assumed.

A.1 Terminology:

Network: A collection of units interconnected in a specific pattern of connectivity. A subset of units receive inputs from external sources, and the network generates an output as a result of the collective processing done by the individual units and interactions between them. The interactions between the units are influenced by the strength of the connections between them. The network undergoes learning by interaction with the environment; the learning process involves a modification of the connection strengths between units.

Unit: Any node in the connectionist network, where all processing is assumed to take place. A unit receives inputs from several neighboring units, combines them in some way to generate an output which is fed to other units in the network. A unit can have several parameters associated with it, which either represent its state or provide information for the computations performed at the unit. Section A.2 contains more information about the *Unit* class.

Site: A location at the unit where connections from neighboring units are visualized as arriving. There can be several sites attached to an unit; the simulator system represents these as a linked-list attached to the unit. All connections arriving at a site are treated alike in computing the net input at the site. Provision of multiple sites allows differential treatment of the inputs. Section A.2 provides more information on the *Site* class.

Link: A connection between two units in the network; connects the output of the source unit to a site at the destination

unit. The simulator system represents the incoming links at a site as a linked list attached to the site. Each link has a *weight* which specifies the strength of the connection. Section A.2 gives more details about the *Link* class.

Activation Functions: Each object (units, sites and links) in the network is assumed to have an activation function associated with it, which represents its action in the network. The link-, site- and unit activation functions are invoked, in that order, during simulation to *update* an unit.

Simulation: The simulation of a network is assumed to take place in *steps*. In each *step*, the input pattern is applied to the input units, the units in the network are updated, and the output generated at the output units is compared with that of the target units (only in case of *supervised learning*) and the weights of the links in the network are updated according to a specific learning rule. Within each step, there may be several *iterations*; in each iteration, the units undergo updation.

A.2 The Unit, Site and Link Classes:

The definitions of the *Unit*, *Site* and *Link* classes, with the data members in the private section of each class, and the public member functions are shown in Tables A.1, A.2 and A.3 respectively. The access to data members in the private section of a class is only through the member functions of the class. Apart from providing (controlled) access, the member functions typically provide means for performing common operations involving the object, such as initializing or displaying it. External functions or member functions of another class can be provided unhindered access to the private members of a class by declaring these to be *friends* of the class, as is the case with *stepSync* and *stepAsync* functions, which have been declared friends of all the three classes. (These functions are the lowest level functions involved in simulation and, as such, there is good reason to provide them unrestricted access for considerations of efficiency.) It may also be noted that the *Unit* class has been provided uncontrolled access to objects of both *Site* and *Link* classes, and the *Site* class to objects of the *Link* class, for similar reasons.

Type *FLINT* will be defined as *int* or *float* during compilation, depending on whether integer version or floating point version of the simulator is specified.

User functions written in C++ can directly make use of the member functions provided in each class for accessing the parameters associated with each unit. However, direct use of the other member functions, such as those for addition or deletion of sites and links, is not advised. Rather, higher level functions provided for the purpose should be used, since they incorporate several checks.

Functions equivalent in action to these member functions, and compatible with C syntax have also been provided, and described later in this manual. They can be used from within user functions

written in C or C++. (User functions written in C should, preferably, conform to the ANSI standard. The C++ compiler that compiles these functions will, otherwise, issue a series of warning messages.)

```
class Unit{
     char  *name;                  // unit name, points to
                                   // "***NO  NAME***"  if
                                  unnamed
     char *type;                   // unit type name
     short init_state;             // unit state after a reset
     short state;                  // unit state
     FLINT init_pot;               // unit potential after a reset
     FLINT pot;                    // unit potential
     FLINT output;                 // unit output
     Site *sites;                  // linked list of attached sites
     short no_sites;               // number of attached sites
     linkTo *fanOut;               // linked list of unit-site pairs
                                   //getting input from the unit -
                                   //fan-out list.
     FLINT data;                   // general-purpose data
     unsigned flags;               // bit vector,  gives  information
                                   // about the unit
      unsigned sets;               // bit vector for set membership
                                   //of  the unit
     ufunc_ptr UnitFunc;           // pointer to unit activation function
  public:
     void setOut(FLINT);           // set unit ouput to specified value
     void setPot(FLINT);           // set unit potential to specified value
     void setState(int);           // set unit state to specified value
     void setData(FLINT);          // set unit data to specified value
     void setFanOut(linkTo *); // set pointer to fan-out list
     void setUnitFunc(ufunc_ptr); // set unit activation function
     char *getName();              // get unit name
     char *getType();              // get unit type name
     getIState();                  // get init_state
     getState();                   // get unit state
     FLINT getIPot();              // get init_potential
     FLINT getPot();               // get unit potential
     FLINT getOut();               // get unit output
     Site * getSites();            // get pointer to list of sites
     getNoSites();                 // get number of attached sites
     linkTo * getFanOut();         // get pointer to fan-out list
     FLINT getData();              // get unit data
     ufunc_ptr  getUnitFunc();       // get pointer to unit activation
                                     //function
     void flag(int);               // set the specified flag (bit)
     void unFlag(int);             // reset the specified flag
     int testFlag(int);            // return TRUE if flag set, else FALSE
     void set(int);                // mark unit as member of given set
     void unSet(int);              // mark unit no more member of given set
     int testSet(int);             // return TRUE if  unit  is  member  of
                                   // given set, else FALSE
```

Table A.1 (contd.)

```
    void initialise(char *,int,int,FLINT,FLINT, FLINT,ufunc_ptr,
                    FLINT =0);
                        // initialise unit parameters

    void display();          // display unit
    Site * getSitePtr(char *);    // get pointer to  a  given
                                  // attached site
    Site * addSite(char *,sfunc_ptr,FLINT); // add site  with
                                  // given initial parameters
    int deleteSite(char *);  // delete given attached site
    void reset();            // set  pot=init_pot,  state=init_state,
                             // output=0
    friend void CopyUnit(Unit *, Unit *);
    friend void NameUnit(char *,int,int,int=0,int=0);
    friend void stepSync();
    friend void stepAsync();
};
```

Table A.1

```
class Site{
    char *name;              // site name
    FLINT value;             // net input at the site
    FLINT data;              // general-purpose data
    Link *links;             // linked list of incoming links
    short no_links;          // number of incoming links
    sfunc_ptr SiteFunc;      // site activation function
    friend Unit;
public:
    Site *next;              // pointer to next site on linked list
     char * getName();       // get site name
    FLINT getValue();        // get site value (net input)
    void setValue(FLINT);    // set site value
    FLINT getData();         // get site data
    void setData(FLINT);     // set site data
    Link * getLinks();       // get pointer to list of attached links
    int getNoLinks();        // get number of attached links
    sfunc_ptr getSiteFunc(); // get pointer to site
                             // activation function
    void setSiteFunc(sfunc_ptr); // set site activation function
    void  initialise(char  *,sfunc_ptr,FLINT);
                             //  initialise   site parameters
    void deletesite();       // delete this site object (itself)
    void display();          // display site
    Link * addLink(int,FLINT,FLINT,lfunc_ptr);
                             // add link with given initial parameters
    Link * getLinkPtr(int);  // get pointer to link from
                             //given unit (index)
    int deleteLink(int);     // delete link from given unit (index)
    friend void stepSync();
    friend void stepAsync();
    friend FLINT SiteValue(char *, Site *sp);
};
```

Table A.2

```
class Link{
     int fromUnit;              // index of source unit
     FLINT weight;              // link weight
     FLINT *value;              // output of unit at other end of link
     FLINT data;                // general-purpose link data
     lfunc_ptr LinkFunc;        // pointer to link activation function
     friend Site;
 public:
     Link *next;                // pointer to next link on the linked  list
     int getFromUnit();         // get index of source unit
     FLINT getWeight();         // get link weight
     void setWeight(FLINT);     // set link weight
     FLINT getValue();          // get output of unit at other end
     FLINT getData();           // get link data
     void setData(FLINT);       // set link data
     lfunc_ptr getLinkFunc(); // get pointer to link activation function
     void setLinkFunc(lfunc_ptr); // set link activation function
     void initialise(int,FLINT,FLINT,lfunc_ptr); // initialise link
     void display();            // display link
     friend void stepSync();
     friend void stepAsync();
 };
```

Table A.3

A.3 ConnectSim++ Commands:

The commands provided by the system for the various tasks have been described below under the respective headings. The commands are typed against the simulator prompt, "-> ", or the break mode prompt, "# ". (A sequence of) Commands may also be stored in files, exactly as they are typed at the prompt, and executed as a batch. The arguments to the commands are typed following the name of the command, and separated by blanks, on the command line. A command is terminated by typing the *<RETURN>* key.

Names of units, unit types, sites, states, sets and functions should not be enclosed in quotes, when used in the command line. An argument representing an unit can be specified either as the index of the unit or by its name. A group of units can be specified as an argument either as a range of consecutive units or a *set* of units. A range of units is represented by specifying the first and last units in the range separated by a '-' with a mandatory space on either side of '-'. (Again, unit indices or names, or a combination of these, can be used to specify the first and last units.) A set of units is specified by the name of the set.

Typing *Ctrl-C* anywhere within the simulator returns you to the simulator prompt.

A.3.1 Help Command:

* *help [<command_name>]*
  The list of commands supported by the system and on-line help for the individual commands, which includes the syntax of the command and a description of the command, are accessible from the command interface with the *help* command. Simply typing *help* without any arguments causes the list of commands, together with the abbreviations, to be displayed. More information about the commands can be obtained by typing *help* followed by the name of the command.

A.3.2 Network Construction Commands:

* *allocateunits [<num>]*
  Creates data space for *<num>* units. May be used at any stage of network construction to add more space. *allocateunits* is automatically invoked by the simulator when it has run out of the space so created by a previous call, or if *makeunit* is invoked prior to invoking *allocateunits*. The indices of the allocated units range from *x+1* to *x+<num>* where *x* is the number of units allocated prior to this call.

* *makeunit <num> <type> <init_state> <state> <init_pot> <pot> <data> <unit_func> <output>*
  Creates *<num>* units of type *<type>* (can be optionally *NULL*). *<type>*, if specified, should be a name that is not already in use. *<init_state>* and *<state>* are integers. *<unit_func>* is the name of the *Unit* Activation Function (which can be NULL if no function is being specified) and the other parameters are integers or floats depending on the simulator type.

* *nameunit <name> < SCALAR / VECTOR / ARRAY > <index> <cols> <rows>*
  Names one or more units. *<index>* gives the first unit which is given the *<name>*. If option *SCALAR* is used, *<name>* is applied to a single unit. For option *VECTOR*, the name is applied to *<cols>* units starting at *<index>* and, for *ARRAY* *<name>* is applied to *<cols>* * *<rows>* units starting at *<index>*.

* *addsite <Unit_Id> <site_name> <site_func> <data>*
  Adds site with name *<site_name>* to the unit(s) specified by *<Unit_Id>*. *<site_func>* gives the site function for the site being added. It can be *NULL* if there is no site function for the added site. *<data>* gives the value with which to initialize the data field of the site.

* *delsite <Unit_Id> <site_name>*
  Deletes the site given by *<site_name>* at the unit(s) specified by *<Unit_Id>*. *Links* arriving at the site(s) are also deleted. If *all* is specified for *<site_name>*, all sites on the unit(s) are deleted.

* *makelink <From_Unit_Id> <To_Unit_Id> <site_name> <weight> <data> <link_func>*
  Sets up a link from the unit(s) given by *<From_Unit_Id>* to the site *<site_name>* at the unit(s) *<To_Unit_Id>*. *<site_name>*

can be *all* to mean that a link is to be made to all sites at unit(s) *<To_Unit_Id>*. The parameters *<weight>*, *<data>*, *<link_func>* give, respectively, the values to initialize the *weight*, *data* and the link function fields with. The *<link_func>* field may be specified as *NULL* which is taken to mean that there is no link function associated with the link being created.

* *deletelink <From_Unit_Id> <To_Unit_Id> <site_name>*
Deletes link(s) from the unit(s) given by *<From_Unit_Id>* to the site(s) indicated by *<site_name>* at the unit(s) given by *<To_Unit_Id>*. *<site_name>* may be *all* meaning all sites at the destination unit. To delete all the links, *deletelinks all all all* may be used.

NOTE:If links do not exist from the unit(s) given by *<From_Unit_Id>* to the specified sites the command silently returns without performing any changes.

* *setoutput <Unit_Id> <value>*
Sets the output of the specified unit(s).

* *setpotential <Unit_Id> <value>*
Sets the potential of the specified unit(s).

* *setstate <Unit_Id> <value>*
Sets the state of the specified unit(s).

* *setlinkweight <From_Unit_Id> <To_Unit_Id> <site_name> <value / random [<mean> <deviation>]>*
Sets the weight of the link from unit(s) given by *<From_Unit_Id>* to *<site_name>* at units specified by *<To_Unit_Id>*. *<site_name>* may be specified as *all* to mean all sites at the destination unit(s). The value for the weight may be specified in one of three ways: *<value>* given explicitly or simply as *random* in which case the weight value will be a random value between Ø and 2 ** 15 - 1 or as *random <low> <high>* which assigns the weight a random value between the limits *<low>* and *<high>*.

NOTE: If links do not exist from the unit(s) given by *<From_Unit_Id>* to the specified sites the command silently returns without performing any changes.

* *setunitfunc <Unit_Id> <function_name / ON / OFF>*
Sets the *Unit* Function of the specified unit(s) to be the function given by *<function_name>*. The *NO_UNITFUNC_FLAG* is turned *OFF* (*ON*) by specifying the option *ON* (*OFF*).

* *setsitefunc <Unit_Id> <site_name> <function_name / ON / OFF>*
Sets the *Site* Function of the site(s) *<site_name>* at the specified unit(s) to be the function given by *<function_name>*. The *NO_SITEFUNC_FLAG* of the unit(s) given by *<Unit_Id>* may be set (unset) by specifying the option *OFF* (*ON*).

* *setlinkfunc <From_Unit_Id> <To_Unit_Id> <site_name> <function_name / ON / OFF>*
Sets the link function of the links from the unit(s) *<From_Unit_Id>* to the site(s) *<site_name>* at the units

<To_Unit_Id> to be the function given by *<function_name>*. The *<site_name>* argument may be *all* to mean all sites at the destination units. The *NO_LINKFUNC_FLAG* of the unit(s) given by *<To_Unit_Id>* may be set (unset) by specifying the option *OFF* (*ON*).

NOTE: If links do not exist from the unit(s) given by *<From_Unit_Id>* to the specified sites the command silently returns without performing any changes.

A.3.3 Set Operation Commands:

* *addtoset <set_name> <Unit_Id>*
  Adds the unit(s) given by *<Unit_Id>* to the set whose name is given by *<set_name>*. If the set does not already exist, it is created.

* *remfromoset <set_name> <Unit_Id>*
  Removes the unit(s) given by *<Unit_Id>* from the set whose name is given by *<set_name>*.

* *deleteset <set_name>*
  Deletes the set whose name is given by *<set_name>*. All units in the set are removed from the set and the name is reset to be unused.

* *unionset <answer_set> <set_1> <set_2>*
  Assigns the union of the sets *<set_1>* and *<set_2>* to the set *<answer_set>*. If a set by name *<answer_set>* exists already and is non-empty, the command fails; if the set does not exist already, it is created.

* *intersectset <answer_set> <set_1> <set_2>*
  Assigns the intersection of the sets *<set_1>* and *<set_2>* to the set *<answer_set>*. If a set by name *<answer_set>* exists already and is non-empty, the command fails; if the set does not exist already, it is created.

* *diffset <answer_set> <set_1> <set_2>*
  Assigns the difference of the sets *<set_1>* and *<set_2>* to the set *<answer_set>*. If a set by name *<answer_set>* exists already and is non-empty, the command fails; if the set does not exist already, it is created.

* *inverseset <answer_set> <set_name>*
  Assigns the inverse of the set <set_name> to the set *<answer_set>*. If a set by name *<answer_set>* exists already and is non- empty, the command fails; if the set does not exist already, it is created.

* *displayset <set_name>*
  Displays the members of the set *<set_name>*. *<set_name>* can be *all* to mean all currently existing sets.

* *memberofsets <Unit_Id>*
  Displays the names of all the sets of which the unit specified by *<Unit_Id>* is a member. For this command, *<Unit_Id>* is restricted so as to specify a single unit -

either by index or by name.

A.3.4 Simulation Commands:

* *sync*
  Sets the simulation mode to be synchronous. This is also the default setting for the update protocol.

* *async [<seed>]*
  Sets the simulation mode to be asynchronous. If *<seed>* is specified, this is used to seed the random number generator.

* *setinputfunc <func_name>*
  Sets the network input function to the name given by *<func_name>*. *<func_name>* has to be a function of type *void*.

* *setlearnfunc <func_name>*
  Sets the network learning function to the name given by *<func_ name>*. *<func_name>* has to be a function of type *void*.

* *setuserfunc <func_name>*
  Sets the general-purpose user-specified function. that is executed once in the number of steps given by *EchoStep*, to the name given by *<func_name>*. (*EchoStep* is the global variable set by the *echo* command.) *<func_name>* has to be a function of type *void*.

* *simulate <step> <iter>*
  Runs the simulation for *<step>* steps each of *<iter>* iterations. The time taken to run the simulation is displayed after the completion.

* *simulate <step> <err>*
  Runs the simulation for *<step>* steps, with as many iterations in each step as to reduce the iteration error to less than or equal to the specified value of *<err>*. The value of the iteration error is set by one of the user specified functions. The time taken to run the simulation is displayed after the completion.

* *test <step> <iter>*
  Tests the performance of the network for *<step>* steps each of *<iter>* iterations. The behavior of the command is exactly identical to that of the *simulate* command except that the learn function is not invoked at the end of each step. The time taken to complete the test is displayed after the completion.

* *test <step> <err>*
  Tests the performance of the network for *<step>* steps, with as many iterations in each step as to reduce the iteration error to less than or equal to the specified value of *<err>*. The value of the iteration error is set by one of the user specified functions. The behavior of the command is exactly identical to that of the *simulate* command except that the learn function is not invoked at the end of each step. The time taken to run the simulation is displayed after the completion.

* *break <step>*
  Suspends simulation after *<step>* steps and returns control to the user. The break mode prompt is displayed by the system. All commands that are valid at the top level are also valid here. The simulation may be resumed with the *continue* command.

* *continue <step>*
  Causes simulation to be resumed after a break. The next break in the simulation is specified with the *<step>* argument, causing the simulation to break after *<step>* steps.

* *do*
  Tells the system that a sequence of commands, to be automatically executed during future breaks in the current simulation, are to follow. The end of the sequence is indicated with the *end* keyword.

* *echo < ON / OFF / StepCount >*
  Sets whether, and how often, the simulator displays the number of steps of simulation completed so far. With option *ON* the message is displayed after every step; if an integer is specified as *StepCount*, the number of steps completed is displayed once in that many steps. The message can be altogether suppressed by specifying the option *OFF*.

A.3.5 Network Examination Commands:

* *displayunit <Unit_Id>*
  Displays all information associated with the unit(s) given by *<Unit_Id>*, including values associated with related sites and links.

* *displaysite <Unit_Id> <site_name>*
  Displays all information associated with the site(s) given by *<site_name>* at the unit(s) given by *<Unit_Id>*, including values associated with links arriving at the site(s).

* *listunit <Unit_Id>*
  Displays summary information associated with the unit(s) given by *<Unit_Id>*. Values associated with related sites and links are not displayed.

* *listsite <Unit_Id> <site_name>*
  Displays summary information associated with the site(s) given by *<site_name>* at the unit(s) given by *<Unit_Id>*. Values associated with links arriving at the site(s) are not displayed. *<site_name>* may be specified as *all* to mean all sites at the given unit(s).

* *listlink(s) to <Unit_Id> <site_name>*
  Displays link(s) to *<Unit_Id> <site_name>* Displays information associated with the link(s) arriving at the site(s) given by *<site_name>* at the unit(s) given by *<Unit_Id>*. *<site_name>* may be specified as *all* to mean all sites at the given unit(s). The commands *list* and *display*

produce identical outputs when links are being displayed.

* *listlink(s) from <Unit_Id>*
  Displays information associated with the link(s) originating at the unit(s) given by *<Unit_Id>*. The commands *list* and *display* produce identical outputs when links are being displayed.

* *listlink(s) <From_Unit_Id> <To_Unit_Id> <site_name>*
  Displays information associated with the link(s) from unit(s) given by *<From_Unit_Id>* to the site(s) given by *<site_name>* at the unit(s) given by *<To_Unit_Id>*. *<site_name>* may be specified as *all* to mean all sites at the destination unit(s). The commands *list* and *display* produce identical outputs when links are being displayed.

* *list [ON / OFF]*
  list step <value>
  list <Unit_Id> < ON / OFF >
  Used to control information that is displayed during simulation. Only summary information about units is displayed. Values associated with related sites and links are not displayed. Without any options, the command causes the present status to be displayed. Listing is turned on or off with *list on* or *list off*. Normally, the units are selected for listing dynamically as the simulation proceeds by one of the unit updation functions (e.g. the Unit Activation Function) which sets the *LIST_FLAG* of the selected unit *ON*. However, units may be preselected for listing by *list <Unit_Id> ON*, where *<Unit_Id>* specifies the units to be listed. Similarly, the *OFF* option deselects the units. Specifying the *step* option results in the units selected to be listed once in every *n* steps, *n* being the value specified.

* *show [ON / OFF]*
  show < step / pot > <value>
  show +/- <Unit_Id>
  show set +/- <set_name>
  Used to control what information is displayed during simulation. Without any options, the command causes the present status to be displayed. Showing is turned on or off with *show on* or *show off*. Specifying the *step* option results in the units selected showing to be displayed every *n* steps, *n* being the value specified. The units to be displayed are selected in one of three ways:

  - Specifying a value for potential with the *pot* option; all units with potential above this value will be shown.
  - Explicitly specifying the unit(s) to be displayed with the + *<Unit_Id>* option (specifying '-' instead of '+' deselects the unit(s));
  - An entire set of units may be selected (deselected) for showing with the *set + (-) <set_name>* option.

* *pipe [< ON / OFF / command >]*
  Feeds output of display and list commands to a *pipe*, rather than directly onto the screen. The *ON* option causes all

future output to be redirected into the *pipe* and sets the *pipe* to be the system command *more*, if no other command has been explicitly specified previously. The *OFF* option resumes direct output on to the screen; specifying a command causes the *pipe* to be set to the specified command, rather than the default *more*. However for the *pipe* to become active, the *pipe* should be switched *ON*. Without any arguments, the command displays the current status.

* *pause [< ON / OFF >]*
  Specifying the *ON* option for the command causes the simulator to wait for user indication to continue after every *show*; the *OFF* option switches off this facility. Without any arguments, the current status is displayed.

A.3.6 File and Miscellaneous Commands:

* *read <cmd_file>*
  Reads and executes commands to build, simulate and display from a command file given by *<cmd_file>* rather than from the keyboard. The command file can also be a *log file* created in a previous session.

NOTE: The *read* command cannot be nested i.e. a command file cannot contain a *read* command.

* *log [on / off]*
  Specifying the *ON* option causes the commands, with the exception of erroneous ones and the *help* commands, to be written to a log file. The name for the log file is initially chosen to be of the form *LOG.xxxxxx*. This can be optionally saved and renamed at the time of exiting the simulator. Switching off the log option with *log OFF* stops all further commands from being logged. The *log* command with no options reports the present status.

* *call <func_name> [<arg1> <arg2>...]*
  Calls from the command interface, the user-specified function given by *<func_name>*. *<arg1>*... are the arguments to the function, which will be collected into a global *argc-argv*-like structure and the function will have to gather its arguments from this structure with the necessary checks. The system does not check in anyway either the number or the type of arguments. The arguments can be integer or floating-point numbers or identifiers. Strings with length not in excess of 25 characters are also accepted. The end of the argument list in the *argv* array is marked by a null string.

* *verbose [on / off]*
  Specifying the *on* option causes diagnostic and warning messages from the lower level routines to be printed; the *off* option results in them being suppressed. With no options, the *verbose* command reports the present status.

* *status*
  Displays the current settings/values of the simulator flags/

parameters.

* *print <string>*
  Prints *<string>* to the standard output. *<string>* is any sequence of characters enclosed in double quotes, barring the double quote character itself.

* *printpause <string>*
  Prints *<string>* to the standard output. *<string>* is any sequence of characters enclosed in double quotes, barring the double quote character itself. After printing the given string it waits for the user to press any key as an indication to continue.

* *whatis <name>*
  Prints the *type* (whether a *Unit* name, set name etc.) of the variable given by *<name>*.

* *clear*
  Causes a clear screen to be displayed.

* *reset*
  Resets the network to a known initial state: potential and state fields are set to their original values and the outputs of all units are reset to zero. No other parameters are modified.

* *restart*
  Cleans out all the simulator data structures so that network construction can begin from scratch, without exiting the simulator.

* *exit*
  Results in a temporary exit to the shell. Typing *exit* at the shell prompt returns control to the simulator.

* *quit*
  Quits the simulator after user confirmation. Before quitting it asks the user if the file containing the log of the commands for the current session should be saved. The user can also specify the name of the log file, or simply type return in which case the simulator itself generates the name of the log file. If the user fails to confirm the *quit* command, the simulator returns to the prompt.

A.3.7 Abbreviations for the commands:

The names of the commands have been so chosen that they clearly reflect the action of the respective commands. However, the user may find it convenient to abbreviate the commands, once he is familiar with the commands and their actions. Also the use of upper-case letters, as shown below, can help improve the readability of the commands, especially within a command file.

The following list provides abbreviations/alternative forms for the commands. Letters enclosed in [] indicate that either is

acceptable and, letter(s) enclosed in parentheses and followed by a ? indicates that these letters are optional and may be omitted to abbreviate the command.

```
[hH](elp)?
[aA](llocate)?[uU](nits?)?
[mM](ake)?[uU](nits?)?
[nN](ame)?[uU](nits?)?
[aA](dd)?[sS](ites?)?
[dD]el(ete)?[sS](ites?)?
[mM](ake)?[lL](inks?)?
[dD]el(ete)?[lL](inks?)?
[sS](et)?[oO](utput)?
[sS](et)?[pP](otential)?
[sS](et)?[sS](tate)?
[sS](et)?[lL](ink)?[wW](eight)?
[sS](et)?[uU](nit)?[fF](unc)?
[sS](et)?[sS](ite)?[fF](unc)?
[sS](et)?[lL](ink)?[fF](unc)?
[rR]eset
[rR]estart
[aA](dd[tT]o)?[sS]et
[rR](em[fF]rom)?[sS]et
[dD]el(ete)?[sS]et
[uU](nion)?[sS]ets?
[iI]nter(sect)?[sS]ets?
[iI]nverse[sS]et
[dD]iff(erence)?[sS]ets?
[dD]isp(lay)?[sS]ets?
[mM]em(ber[oO]f)?[sS]ets?
[sS]ync
[aA]sync
[sS](et)?[iI]n(put)?[fF](unc)?
[sS](et)?[lL]earn[fF](unc)?
[sS]im(ulate)?
[tT]est
[eE]cho
[dD]isp(lay)?[uU](nits?)?
[dD]isp(lay)?[sS](ites?)?
[dD]isp(lay)?[lL](inks?)?
[lL](ist)?[uU](nits?)?
[lL](ist)?[sS](ites?)?
[lL](ist)?[lL](inks?)? [ from : to]
[lL]ist
[sS]how
[pP]ipe
[pP]ause
[sS]tat(us)?
[rR]ead
[lL]og
[cC]all
[pP]r(int)?
[pP](rint)?pause
[wW]hat(is)?
[qQ](uit)?
```

A.4 ConnectSim++ Functions:

The system provides an elaborate set of functions which may be invoked from within user functions written in either C or C++. Functions to implement the operations involved in each phase of the network definition and simulation process exist and are described below under the respective headings.

A.4.1 Network Construction Functions:

* *void AllocateUnits(int number);*
  Creates space for the specified number of units. Also creates the *Outputs* array (used to store the output values of the units during simulation) of a corresponding size. Can fail if the number specified is too large, and the system is not able to allocate memory of the requested size.

* *int MakeUnit(char *type, int init_state, int state, int init_pot, int pot, int data, ufunc_ptr UnitFunc, int output);*
  Creates an unit with its parameters initialized to values specified as arguments. *UnitFunc* is one of the library functions or a user-specified function or can be specified as *NULL*, in which case the activation function is taken to be the null function. *MakeUnit* returns the index, in the unit array, of the unit created by the call. The first call to *Makeunit* returns index Ø and subsequent calls return consecutive indices.

* *Site *MakeSite(int u_index, char *name, sfunc_ptr SiteFunc, int data);*
  Attaches a site with name *name* to the unit whose index is *u_index*. *SiteFunc* gives the pointer to the site activation function which can be user-specified, a library- or a null function. *data* gives the value with which to initialize the *data* field of the site. The call returns a pointer to the newly attached site.

* *Link *MakeLink(int from, int to, char *site_name, int weight, int data, lfunc_ptr LinkFunc);*
  Sets up a link from the unit with index *from* to the site given by *site_name* at the unit with index *to*. *weight* and *data* specify the initial values for the weight of the connection and the link *data* field respectively. *LinkFunc* is a pointer to the link activation function. The function returns a pointer to the link that is set up.

* *void NameUnit(char *name, int type, int index, int cols, int rows);*
  Assigns the specified *name* to a single unit or a vector or a 2-D array of units. *name* is the pointer to the character string giving the *name*, and *type* specifies the type of the *name* - *SCALAR* (for a single unit), *VECTOR* (for a 1-D array or vector of units) or *ARRAY* (for a 2-D array of units). *index* gives the index, in the unit array, of the unit to be named or of the first unit in the vector or array. The argument *cols* need be specified only if the type is *VECTOR* or *ARRAY*. Similarly, *rows* is specified only if the type is *ARRAY*. The

name has the form *name[c]* if the name is of type *VECTOR*, and *name[r][c]* if it is of type *ARRAY*, where *c* and *r* denote the indices. *cols* (*rows* * *cols*) consecutive units are given the specified name, if the name is of type *VECTOR* (*ARRAY*).

* *DeclareState(char *name, int state);*
  Associates the name given by *name* with the state *state*.

* *void DeleteSite(int u_index, char *site_name);*
  Deletes the site with name *site_name* attached to the unit whose index is given by *u_index*.

* *void DeleteLink(int from, int to, char *site_name);*
  Removes the link from unit with index *from* to the site *site_name* at unit with index *to*. If *site_name* is specified to be *all* or *ALL*, the link originating at unit *from* and arriving at each site at the unit *to* (if such a link exists), is removed.

A.4.2 Functions for Set Operations:

Functions are provided for all essential set-theoretic operations. All functions, except *MemberSet()*, *IsSet()* and *MemberOfSets()*, return a value of -1 if the call fails.

* *int DeclareSet(char *name);*
  Creates a set, which is initially empty, with the specified name.

* *int AddToSet(char *name, int low, int high);*
  Adds units with indices *low* through *high* to the given set.

* *int RemFromSet(char *name, int low, int high);*
  Removes units with indices *low* through *high* from the given set.

* *int UnionSet(char *name3, char *name1, char *name2);*
  Assigns the union of sets *name1* and *name2* to the set *name3*.Creates set *name3* if it does not exist.

* *int IntersectSet(char *name3, char *name1, char *name2);*
  Assigns the intersection of sets *name1* and *name2* to the set *name3*. Creates set *name3* if it does not exist.

* *int DifferenceSet(char *name3, char *name1, char *name2);*
  Assigns the difference of sets *name1* and *name2* (i.e. those elements in set *name1* but not in set *name2*) to the set *name3*. Creates set *name3* if it does not exist.

* *int InverseSet(char *name2, char *name1);*
  Assigns to set *name2* all units not in set *name1*. Creates set *name2* if it does not exist.

* *int DeleteSet(char *name);*
  Removes the set *name* from the system.

* *int DisplaySet(char *name);*
  Displays the indices of the units which are members of the set *name*.

* *int MemberSet(char *name, int unit_index);*
  Returns *TRUE* if the unit given by *unit_index* is a member of the set *name*, else returns *FALSE*.

* *int MemberOfSets(int unit_index);*
  Displays the names of all sets of which the unit given by *unit_index* is a member.

* *int IsSet(char *name);*
  Returns *TRUE* if the name *name* is that of a set, *FALSE* otherwise.

A.4.3 Functions for Accessing Unit, Site, and Link Parameters:

These functions enable the information associated with the unit, site and link objects to be read or modified.

(i) Functions for the Unit Class:
Note: Function names that end with a *P* access the unit object through a pointer to it.

* *void SetOut(int i,FLINT o);*
  Sets *output* of unit with index *i* to the value *o*.

* *void SetOutP(Unit *up,FLINT o);*
  Sets *output* of unit pointed at by *up* to the value *o*.

* *void SetPot(int i,FLINT p);*
  Sets *pot* (potential) of unit with index *i* to the value *p*.

* *void SetPotP(Unit *up,FLINT p);*
  Sets *pot* of unit pointed at by *up* to the value *p*.

* *void SetState(int i,int s);*
  Sets *state* of unit with index *i* to the value *s*.

* *void SetStateP(Unit *up,int s);*
  Sets *state* of unit pointed at by *up* to the value *s*.

* *void SetData(int i,FLINT d);*
  Sets *data* of unit with index *i* to the value *d*.

* *void SetDataP(Unit *up,FLINT d);*
  Sets *data* of unit pointed at by *up* to the value *d*.

* *void SetFanOut(int i,linkTo *lt);*
  Sets *fanOut* (the pointer to the fanout-list) of unit with index *i* to *lt*.

* *void SetFanOutP(Unit *up,linkTo *lt);*
  Sets *fanOut* (the pointer to the fanout-list) of unit pointed at by *up* to *lt*.

* *char * GetUnitName(int i);*
  Returns (pointer to) *name* of unit with index *i*.

* *char * GetUnitNameP(Unit *up);*
  Returns *name* of unit pointed at by *up*.

* *char * GetUnitType(int i);*
  Returns *type* of unit with index *i*.

* *char * GetUnitTypeP(Unit *up);*
  Returns *type* of unit pointed at by *up*.

* *int GetInitState(int i);*
  Returns *init_state* (state value following a *reset*) of unit with index *i*.

* *int GetInitStateP(Unit *up);*
  Returns *init_state* of unit pointed at by *up*.

* *int GetState(int i);*
  Returns *state* of unit with index *i*.

* *int GetStateP(Unit *up);*
  Returns *state* of unit pointed at by *up*.

* *FLINT GetInitPot(int i);*
  Returns *init_pot* (state value following a *reset*) of unit with index *i*.

* *FLINT GetInitPotP(Unit *up);*
  Returns *init_pot* of unit pointed at by *up*.

* *FLINT GetPot(int i);*
  Returns *pot* of unit with index *i*.

* *FLINT GetPotP(Unit *up);*
  Returns *pot* (potential) of unit pointed at by *up*.

* *FLINT GetOut(int i);*
  Returns *output* of unit with index *i*.

* *FLINT GetOutP(Unit *up);*
  Returns *output* of unit pointed at by *up*.

* *Site * GetSites(int i);*
  Returns *sites* (pointer to the first site on the linked list of attached sites) of unit with index *i*.

* *Site * GetSitesP(Unit *up);*
  Returns *sites* of unit pointed at by *up*.

* *int GetNumSites(int i);*
  Returns *no_sites* (number of sites) of unit with index *i*.

* *int GetNumSitesP(Unit *up);*
  Returns *no_sites* of unit pointed at by *up*.

* *linkTo * GetFanOut(int i);*
  Returns *fanOut* of unit with index *i*.

* *linkTo * GetFanOutP(Unit *up);*
  Returns *fanOut* of unit pointed at by *up*.

* *FLINT GetData(int i);*
  Returns *data* of unit with index *i*.

* *FLINT GetDataP(Unit *up);*
  Returns *data* of unit pointed at by *up*.

* *ufunc_ptr GetUnitFunc(int i);*
  Returns *UnitFunc* (pointer to the unit activation function) of unit with index *i*.

* *ufunc_ptr GetUnitFuncP(Unit *up);*
  Returns *UnitFunc* of unit pointed at by *up*.

* *void SetFlag(int i,int n);*
  Sets flag *n* (bit position *n* of *flags*) of unit *i*, to 1.

* *void SetFlagP(Unit *up,int n);*
  Sets flag *n* (bit position *n* of *flags*) of unit pointed at by *up*, to 1.

* *void ResetFlag(int i,int n);*
  Sets flag *n* (bit position *n* of *flags*) of unit *i*, to Ø.

* *void ResetFlagP(Unit *up,int n);*
  Sets flag *n* (bit position *n* of *flags*) of unit pointed at by *up*, to Ø.

* *void TestFlag(int i,int n);*
  Returns *TRUE* if flag *n* of unit *i* is set to 1, *FALSE* otherwise.

* *void TestFlagP(Unit *up,int n);*
  Returns *TRUE* if flag *n* of unit pointed at by *up* is set to 1, *FALSE* otherwise.

* *Site * GetSitePtr(int i,char *n);*
  Returns pointer to site *n* of unit *i*. Returns pointer to first site in linked list if second argument is not specified.

* *Site * GetSitePtrP(Unit *up,char *n);*
  Returns pointer to site *n* of unit pointed at by *up*. Returns pointer to first site in linked list if second argument is not specified.

(ii) Functions for the Site Class:

* *void SetSiteValue(Site *sp, FLINT v);*
  Sets *value* of the site pointed at by *sp* to the value *v*.

* *void SetSiteData(Site *sp, FLINT d);*
  Sets *data* of the site pointed at by *sp* to the value *d*.

* *void SetSiteFunc(Site *sp, sfunc_ptr fp);*
  Sets *SiteFunc* (pointer to the site activation function) of the site pointed at by *sp* to the value *fp*.

* *char * GetSiteName(Site *sp);*
  Returns (pointer to) *name* of site pointed at by *sp*.

* *FLINT GetSiteValue(Site *sp);*
  Returns *value* (the net input from all incoming links) of site pointed at by *sp*.

* *FLINT GetSiteData(Site *sp);*
  Returns *data* of site pointed at by *sp*.

* *Link * GetLinks(Site *sp);*
  Returns *links* (pointer to the first link in the list of attached links) of site pointed at by *sp*.

* *int GetNumLinks(Site *sp);*
  Returns *no_links* (number of incoming links) of site pointed at by *sp*.

* *sfunc_ptr GetSiteFunc(Site *sp);*
  Returns *SiteFunc* of site pointed at by *sp*.

* *Link * GetLinkPtr(Site *sp, int f);*Returns pointer to link from unit *f* at site pointed at by *sp*.

(iii) Functions for the Link Class:

* *void SetWeight(Link *lp, FLINT w);*
  Sets *weight* of link pointed at by *lp* to value *w*.

* *void SetLinkData(Link *lp, FLINT d);*
  Sets *data* of link pointed at by *lp* to value *d*.

* *void SetLinkFunc(Link *lp, lfunc_ptr fp);*
  Sets *LinkFunc* (pointer to link activation function) of link pointed at by *lp* to value *fp*.

* *int GetFromUnit(Link *lp);*
  Returns the index of the unit where the link originates.

* *FLINT GetWeight(Link *lp);*
  Returns *weight* of link pointed at by *lp*.

* *FLINT GetLinkValue(Link *lp);*
  Returns *value* (output of the unit at the other end of the link, *fromUnit*) of site pointed at by *sp*.

* *FLINT GetLinkData(Link *lp);*
  Returns *data* of link pointed at by *lp*.

* *lfunc_ptr GetLinkFunc(Link *lp);*
  Returns *LinkFunc* (pointer to link activation function) of link pointed at by *lp*.

A.4.4 Functions for Examining the Network:

In the following functions, it is possible to specify *all* (without any enclosing quotes) as an argument in place of *low*; in this case, the argument *high* is not specified. The range of the units is taken to be all the existing units. Also, if a single a unit is to be displayed, either the same value (the index of the unit to be displayed) is provided in place of both *low* and *high*, or the index of the unit is specified as the value for *low* and the argument *high* is omitted.

* *DisplayUnit(int low, int high);*
  Displays the units with indices in the range *low* through *high*.

* *ListUnit(int low, int high);*
  Displays the units with indices in the range *low* through *high* in compact form.

* *DisplayUnitSet(char *name);*
  Displays the units which are members of the set specified by *name*.

* *ListUnitSet(char *name);*
  Displays the units which are members of the set specified by *name* in compact form.

* *DisplaySite(int low, int high, char *name);*
  Displays the sites with name specified by *name* and attached to units with indices in the range *low* through *high*. If a site of that name does not exist at some unit in the specified range, the function return silently. If *all* or *ALL* is specified in place of *name* all sites attached to the units are displayed.

* *ListSite(int low, int high, char *name);*
  Displays the sites with name specified by *name* and attached to units with indices in the range *low* through *high* in compact form. If *all* or *ALL* are specified in place of *name* all sites attached to the units are displayed. If a site of that name does not exist at some unit in the specified range, the function returns silently. If *all* or *ALL* is specified in place of *name* all sites attached to the units are displayed.

* *DisplayLink(int from, int to, char *name);*
  Displays the link connecting the unit with index *from* to the site *name* at the unit with index *to*. If *all* or *ALL* is specified in place of *name* links from unit *from* arriving at each site of unit *to* are displayed.

* *ListLinksTo(int low, int high, char *name);*
  Displays all the links arriving at the site *name* of the units with indices in the range *low* through *high*. If a site of that

name does not exist at some unit in the specified range, the function returns silently. If *all* or *ALL* is specified in place of *name* all sites attached to the units are displayed.

* *ListLinksFrom(int low, int high);*
  Displays all the links originating at the units with indices in the range *low* through *high*.

* *int PipeSet(char *n);*
  Sets the *pipe* (a *Unix* system command, to which the output of the simulator during a *show* or *list*, is sent) to be the command specified by *n*. If no argument is specified, the *pipe* is set to the *more* command of *Unix*.

* *void PipeBegin();*
  Piping is enabled; all future output is sent to the *pipe* (set with the *PipeSet()* function), rather than directly to the display screen. If the *pipe* has not been set, the default *pipe* command is assumed to be *more*.

* *void PipeEnd();*
  Piping is disabled; all future output is sent directly to the display screen.

* *void PerformShow();*
  If the global variable *Show* is set to 1 (*ON*), displays all the units marked for a *show*, at the end of each step of simulation. A unit is marked for *show* either by setting its *SHOW_FLAG* to 1 or by adding it to the system created set, *ShowSet*.

* *void PerformList();*
  If the global variable *List* is set to 1 (*ON*), displays all the units marked for a *list*, at the end of each step of simulation. A unit is marked for *list* either by setting its *LIST_FLAG* to 1.

* *void pause();*
  Suspends further output to the display screen until the user hits some key on the keyboard.

A.4.5 Functions for Simulation:

Simulation is invariably performed from the command interface, since this generally provides better control over the entire process. (In fact, we strongly recommend this.) However, functions to perform simulation from within a user function are available, and their description is provided below. If these are used, the user function will have to explicitly set the system variables which control the simulation. These variables are described in A.6.

The simulation process consists of the following actions:

(a) Invoke the user-written network input function, for obtaining values that correspond to an input pattern, and clamp these on the input units. (If target units exist, the target pattern is obtained and clamped on the target units.)

(b) Update all units in the network (by calling the unit, site and link activation functions) *iter* number of times. The mode of updation is *asynchronous* if the *SimMode* flag is set to *ASYNC*, *synchronous* otherwise.
(c) Apply the learning rule (by calling the *learn function*) to update the weights of the links.
(d) If the *Show* and *List* flags are set, *show/list* the units marked for display (see description of *show* and *list* commands).

After the simulation the network may be tested for its performance. Testing, essentially consists of the same actions as above, except that in step (a) no inputs are clamped on the target units, and step (c) is not performed.

* *void SimulateIter(int step, int iter);*
Performs simulation of the network for *step* steps, with each step having *iter* iterations. If the network input function and the learning function have not been specified, warns the user.

* *void SimulateErr(int step, FLINT err);*
Performs simulation of the network for *step* steps, with each step having a sufficient number of iterations to reduce the iteration error to below the value specified by *err*. If the network input function and the learning function have not been specified, warns the user.

* *void TestIter(int step, int iter);*
Tests the network for *step* steps, with each step having *iter* iterations. If the network input function has not been specified, warns the user.

* *void TestErr(int step, FLINT err);*
Tests the network for *step* steps, with each step having a sufficient number of iterations to reduce the iteration error to below the value specified by *err*. If the network input function has not been specified, warns the user.

A.4.7 Miscellaneous Functions:

* *Unit * UnitPtr(int i);*
Returns the pointer to the unit whose index in the unit array is *i*.

* *Unit * UnitIndex(Unit *up);*
Returns the index, in the unit array, of unit pointed at by *up*.

* *int randint(int low, int high);*
Returns a random integer in the range *low* - *high*.

* *float rand01();*
Returns a random floating point number in the range 0 - 1.

* *float randf(float low, float high);*
Returns a random floating point number in the range *low* - *high*.

A.4.6 Name Table Access Functions:

The system maintains a hashed name table whose entries have the following structure:

```
typedef struct name_node{
     char *name;          /* string containing the name */
     int nType;           /* type of the name */
     int nIndex;          /* index, explained below */
     int nCols;           /* number of columns */
     int nRows;           /* number of rows */
     name_node *next;     /* pointer to next node */
} NN;
```

The value stored in the *nIndex* field is interpreted according to the valueof *n Type*:
If *nType* = *SCALAR*, *nIndex* stores the index, in the unit array, of the unit represented by *name*.
If *nType* = *VECTOR* or *ARRAY*, *nIndex* stores the index, in the unit array, of the first of the sequence of units, that make up the vector or array, represented by *name*.
If *nType* = *FUNC_SYM*, *nIndex* stores the pointer to the function. (When retrieved, it should be suitably typecast.)
If *nType* = *SET_SYM*, *nIndex* stores the index in the Set Table, of the set represented by *name*.
In all other cases, it is undefined.
nCols and *nRows* are defined, only when *nType* = *VECTOR* or *ARRAY*; represent the number of columns in the vector (*nRows* undefined for *VECTOR*), and the number of columns and rows in the array.
The definitions of the constants for *nType* is given below:

```
/* Definition of Name Table Entry types. */
#define FREE_SYM 0
#define SCALAR 1
#define VECTOR 2
#define ARRAY 3
#define TYPE_SYM 4
#define SITE_SYM 5
#define FUNC_SYM 6
#define SET_SYM 7
#define STATE_SYM 8
#define DATA_SYM 9
```

The system provides a set of functions for accessing the name table and manipulating it. However, user functions very rarely need to use them.

* *NN * EnterName(char *name,int type, int index, int cols, int rows);*
Stores *name* with the given values for the other data. Returns the pointer to the node so created. If name already exists, and the other data stored with it match the values given, simply return the pointer to the node; if the values do not

match return *NULL* (to indicate an error condition).

* *NN * LkpName(char *name);*
  If *name* exists in the name table, returns pointer to the entry (node), *NULL* otherwise.

* *int DeleteName(char *name);*
  Deletes the entry for *name*; returns -1 if name does not exist.

* *char * GetTypeName(int i);*
  Returns a poinetr to a volatile string containing the name of the type specified by *i*.

* *char * NameToType(char *n);*
  Returns a pointer to a volatile string containing the type of the name *n*.

* *int NameToInd(char *name,int cols,int rows);*
  Returns the index, in the unit array, of the unit specified by *name*, *cols* (if *VECTOR* or *ARRAY*), and *rows* (if *ARRAY*). Returns -1 if no such unit exists.

* *char * IndToName(int);*
  Returns the name of the unit, whose index in the unit array is *i*; the name is of the form *name* (for *SCALAR*), *name[c]* (for *VECTOR*) or *name[r][c]* (for *ARRAY*) where *r* and *c* are the row- and column subscripts respectively.

* *int NameToState(char *name);*
  Returns the integer that represents the state having name *name*.

* *char * StateToName(int i);*
  Returns the name that represents the state given by *i*.

* *int GetFuncPtr(char *name);*
  Returns the pointer to the function *name*; must be suitably typecast before use. Returns *NULL* if no such function exists.

* *char *GetFuncName(int fp);*
  Returns the name of the function with pointer value *fp*; returns *NULL* if no such function exists. (The pointer to the function must be typecast to an *int* before passing it as an argument.)

* *void DispNameTab();*
  Display the entire name table.

* *char * NameToType(char *name);*
  Returns a pointer to a volatile string describing the type of *name*; returns a pointer to a null string, if *name* does not exist.

A.4.7 Important Simulator Variables:

Many global variables used by the simulator are accessible to user code. Modifying these should be done with care.

*Unit *u* Pointer to unit array, the main data structure.

*FLINT *Outputs* Vector of unit output values. *Links* get their values with a pointer into this array. Used and updated during simulation.

int MaxIndex The number of units for which space has been allocated so far.

int CurIndex The number of units that actually exist.

NN NT [HTSIZ] The *Name Table* structure. Each element of the array begins a linked list that stores names that hash onto that index. *HTSIZ* is a defined constant, which is generally a prime number.

STNode ST[32] The *Set Table* structure. This array stores the entries for all sets so far created by the system. A maximum of 32 sets can exist at any time. The system creates the *ShowSet* when it is started.

int setCount The number of sets that currently exist in the system.

int FileCmd A boolean telling the system whether commands are currently being read from a file or from the keyboard.

int verbose A boolean to control printing of diagnostic messages by low-level functions. Messages are suppressed if set to Ø, displayed otherwise.

int SimMode A boolean to indicate the simulation update protocol - *SYNC* or *ASYNC*.

int List A boolean to indicate whether *listing* of marked (*LIST_FLAG* = 1) units is to be enabled during simulation.

int ListStep Number of steps of simulation between *listing*.

int Show A boolean indicating whether *showing* is enabled.

*char * ShowSet* Set which contains units to be *shown* during simulation.

unsigned int ShowSets A bit vector indicating sets whose members are marked for *show* in addition to those of *ShowSet*.
FLINT ShowPot During a *show*, display all units with potential higher than this value.

int ShowStep Number of steps of simulation between *showing*.

int Pause A boolean to indicate whether *pause* is enabled.

int Pipe A boolean to indicate whether *pipe*ing is enabled.

*char *PipeCom* Name of *pipe* process to use for display output, if *pipe*ing is enabled.

*FILE * Dispf* The file to which display output is written, normally *stdout*; points to the *pipe* process when *pipe*ing is enabled.

int Echo A boolean indicating whether *echo*ing is enabled.

int EchoStep Number of steps between *echo* messages.

int Log A boolean indicating whether commands typed at the keboard are being saved in the log file.

*FILE * LogFile* The pointer to the file into which the commands are being saved, if *logging* is enabled.

char LogFileName[30] Name of the log file; 30 characters long, maximum.

FLINT IterErr Value of the iteration error during the previous iteration.

*void (*NetInputFunc)(int) NetInputFunc* is the pointer to the network input function.

*void (*LearnFunc)() LearnFunc* is the pointer to the network *learn function.*

int NoSites Number of sites currently existing in the system.

int NoLinks Number of links currently existing in the system.

A.4.8 Important Defined Constants and User-defined Types:

```
/* Definition of Boolean Constants. */
#define TRUE (1)
#define FALSE (0)
#define ON 1
#define OFF 0

/* Unit Flags. */
#define NO_UNITFUNC_FLAG 1
#define NO_SITEFUNC_FLAG 2
#define NO_LINKFUNC_FLAG 3
#define STEP_SIM_FLAG 4
#define LIST_FLAG  5
#define SHOW_FLAG  6

/* The FLINT type */
#ifdef FSIM
#define FLINT float
#else
#define FLINT int
#endif
```

```
/* Simulation Mode */
#define SYNC 1
#define ASYNC 2

#define UNITSINCR 10
/* increment for unit array size on automatic call to
AllocateUnits() */

#define HTSIZ 13 /* Hash Table Size - Better be a prime */

#define all -77 /* argument to pass to a function, to mean all
units */

#define FAIL -1 /* value to return on an unsuccessful
function call */

#define DONE 2 /* value to return on a successful function
call */

/* structure of node in the fan-out list. */
struct link_to{
     int toUnit;
     char *toSite;
     struct link_to *next;
};
typedef struct link_to linkTo;

/* Type definitions for activation functions */
typedef void func_type;
typedef func_type (* ufunc_ptr)(Unit *);
typedef func_type (* sfunc_ptr)(Site *);
typedef func_type (* lfunc_ptr)(Link *);
```

APPENDIX B

Code for Simulations

PATTERN ASSOCIATOR NETWORK

```
/* Functions for simulation of a Pattern Associator Network.  */

#include "defs.h"  // Files containing the definition of the simulator
#include "extern.h"   // data structures and other global variables.
#include "sim.h"
#define ETA 10         // Constant representing the learning rate.
typedef int * intp;
intp *ipat, *tpat;    // Array to store the input and target patterns.
func_type SFweightedSum(Site *);  // Library function for net site input.
FILE *infile;
int unitCount;           // No. of input (also output & target) units
int patCount;                      // No. of patterns.

/* Function for constructing the network. */
void BuildNet(){
int i, j;
void UFpass(Unit *up);
void SFweightedSum(Site *sp);
     if (argc !=3){
          printf("Usage: call BuildNet <file_name>\n");
          return;
     }
// Open the input data file; the name of this file is passed as an
// argument in the global argv array.

     infile = fopen(argv[2],"r");
     if (infile == NULL){
          cout << form("sim: could not open %s\n",argv[2]);
          return;
     }

// Get the number of units and the number of patterns.
     fscanf(infile,"%d %d",&unitCount,&patCount);
     AllocateUnits(3*unitCount);     // Create space for input, output
                                // and target units.

     for (i=0; i<unitCount; i++){          // Make the input  units.
          MakeUnit("INPUT",0,0,0,0,0,NULL,0);
          SetFlag(i,NO_UNITFUNC_FLAG);  // No Unit activation function.
          SetFlag(i,NO_SITEFUNC_FLAG);  // No Site activation function.
          SetFlag(i,NO_LINKFUNC_FLAG);  // No Link activation function.
     }
     NameUnit("In",VECTOR,0,unitCount);  // Name them as a vector - In.
     DeclareSet("InputSet");           // Make a set of the input units.
     AddToSet("InputSet",0,unitCount-1);

     for (i=unitCount; i<2*unitCount; i++){ // Make the output units.
          MakeUnit("OUTPUT",0,0,0,0,0,UFpass,0);
          MakeSite(i,"feed",SFweightedSum,0);
                                // Add a site with name "feed".
          MakeSite(i,"teach",NULL,0);  // Add a site with name "teach".
```

```
          SetFlag(i,NO_LINKFUNC_FLAG);
     }
     NameUnit("Out",VECTOR,unitCount,unitCount);
     DeclareSet("OutputSet");
     AddToSet("OutputSet",unitCount,2*unitCount-1);
     for (i=2*unitCount; i<3*unitCount; i++){ // Make the target units.
          MakeUnit("TEACHER",0,0,0,0,0,NULL,0);
          SetFlag(i,NO_UNITFUNC_FLAG);
          SetFlag(i,NO_SITEFUNC_FLAG);
          SetFlag(i,NO_LINKFUNC_FLAG);
     }
     NameUnit("Teach",VECTOR,2*unitCount,unitCount);
     DeclareSet("TeachSet");
     AddToSet("TeachSet",2*unitCount,3*unitCount-1);

// Make links from input units to site "feed" at
// output units with weight = 0.
     for (i=0; i<unitCount; i++)
          for (j=unitCount; j<2*unitCount; j++)
               MakeLink(i,j,"feed",0,0,NULL);

// Make links from target units to site "teach" at
// output units, weight = 1000
     for (i=unitCount; i<2*unitCount; i++)
          MakeLink(i+unitCount,i,"teach",1000,0,NULL);
}
/* Function for setting the pattern for the input units.
   This C++ function may be easily rewritten for the C compiler by
   using 'malloc' in place of 'new'and 'printf' in place of 'cout <<' */

void ReadPat(int mode){
static int readInput = FALSE;
static int CurPat=0;
int i,j,n;
     if (!readInput){         // To read the input file and create
                              // the arrays only
          readInput = TRUE;    // on the first call to this function.

/****  This has to be done in place of a simple declaration such as:
   int ipat[m][n] because the number of units and patterns will be
   known only at run time.  */

          ipat = new intp[patCount];
          for (i=0; i<patCount; i++)      // Allocate array for
                                          // input patterns...
               ipat[i] = new int[unitCount];
          tpat = new intp[patCount];
          for (i=0; i<patCount; i++)      // ...and the target patterns.
               tpat[i] = new int[unitCount];
/*****/
          for (i=0; i<patCount; i++)      // Read in the input patterns...
               for (j=0; j<unitCount; j++)
                    if (!(n = fscanf(infile,"%d ",&ipat[i][j]))){
                         cout << "Premature end of file.\n";
                         exit(1);
```

```
                    }
          for (i=0; i<patCount; i++)     // ...and the target patterns.
               for (j=0; j<unitCount; j++)
                    if (!(n = fscanf(infile,"%d ",&tpat[i][j]))){
                         cout << "Premature end of file.\n";
                         exit(1);
                    }
// Display the read patterns.
          cout << "Input Pattern Set:\n";
          for (i=0; i<patCount; i++){
               for (j=0; j<unitCount; j++)
                    cout << form("%d ",ipat[i][j]);
               cout << "\n";
          }
          cout << "\n";
          cout << "Target Pattern Set:\n";
          for (i=0; i<patCount; i++){
               for (j=0; j<unitCount; j++)
                    cout << form("%d ",tpat[i][j]);
               cout << "\n";
          }
     }
// All of the above done only once.

// On each call set input vector on input units and
// 0 on each output unit.
     for (j=0; j<unitCount; j++){
          SetOut(j,ipat[CurPat][j]);
          SetOut(j+unitCount,0);
     }

// Set target vector on target units only if called during simulation.
     if (mode == SIM)
          for (j=0; j<unitCount; j++)
               SetOut(j+2*unitCount,tpat[CurPat][j]);
     else
          for (j=0; j<unitCount; j++)
               SetOut(j+2*unitCount,0);
     CurPat++;
     CurPat %= patCount;    // To cycle over the same set of patterns.
}

/* The Learning Rule.    */
void LearnPass(){
Site *sp;
Link *lp;
     for (int i=unitCount; i<2*unitCount; i++)
          if (Outputs[i] != Outputs[i+unitCount]){
               sp = GetSitePtr(i,"feed");
               lp = GetLinks(sp);
               for ( ; lp; lp=lp->next)
                    if (GetLinkValue(lp))
                         if (Outputs[i+unitCount])
                              SetWeight(lp,GetWeight(lp) + ETA);
                         else
```

```
                                        SetWeight(lp,GetWeight(lp) - ETA);
          }
}

/* The Unit Activation function for the output units. */
void UFpass(Unit *up){
Site *sp;
     sp = GetSitePtrP(up,"feed");
     if (GetSiteValue(sp) > 0){
          SetPotP(up,1);
          SetOutP(up,1);
     }

     else{
          SetPotP(up,0);
          SetOutP(up,0);
     }
}

/* Function to print the final values for the weights,
   after simulation. */
void PrintWts(){
Site *sp;
Link *lp;
     cout << "\n";
     cout << "The final values for the weights are:\n";
     for (int j=unitCount; j<2*unitCount; j++){
          sp = GetSitePtr(j,"feed");
          for (lp=GetLinks(sp); lp; lp=lp->next)
               cout << form("%5d ",GetWeight(lp));
          cout << "\n";
     }
}
```

Once the functions are compiled with the simulator code, the simulator may be started, and the simulations run with the following commands. (The list represents a typical sequence of commands. The user may like to include commands for examining the network etc. In most cases, the set and sequence of commands are similar.)

```
 call
BuildNet <data_file>
setinputfunc ReadPat
setlearnfunc  LearnPass
list  all  on
list off
echo  off
pause on
list
echo
show
pause
pipe
```

```
sim 20 5
list  on
test 18 5
call PrintWts
```

AUTO ASSOCIATOR NETWORK

```
/* Functions for simulation of an Auto Associator Network. */

#include "defs.h"
#include "extern.h"
#include "sim.h"
#define ETA 10
typedef int * intp;
intp *ipat, *tpat;
func_type SFweightedSum(Site *);
int count,count2;

void ReadPat(int mode){
FILE *infile;
static int readInput = FALSE;
static int CurPat=0;
int i,j,n;
// Read in input and target patterns from input file.
     if (!readInput){
          readInput = TRUE;
          infile = fopen("autas.dat","r");
          if (infile == NULL){
               cout << form("sim: could not open autas.dat\n");
               return;
          }
          fscanf(infile,"%d %d",&count,&count2);
          ipat = new intp[count2];
          for (i=0; i<count2; i++)
               ipat[i] = new int[count];
          tpat = new intp[count2];
          for (i=0; i<count2; i++)
               tpat[i] = new int[count];
          for (i=0; i<count2; i++)
               for (j=0; j<count; j++)
                    if (!(n = fscanf(infile,"%d ",&ipat[i][j]))){
```

```
                    cout << "Premature end of file.\n";
                    exit(1);
                }
        for (i=0; i<count2; i++)
            for (j=0; j<count; j++)
                if (!(n = fscanf(infile,"%d ",&tpat[i][j]))){
                    cout << "Premature end of file.\n";
                    exit(1);
                }
// Display the read patterns.
        cout << "Input Pattern Set:\n";
        for (i=0; i<count2; i++){
            for (j=0; j<count; j++)
                cout << form("%d ",ipat[i][j]);
            cout << "";
        }
        cout << "\n";
        cout << "Target Pattern Set:\n";
        for (i=0; i<count2; i++){
            for (j=0; j<count; j++)
                cout << form("%d ",tpat[i][j]);
            cout << "";
        }
    }
// All the above done only once.
// Now set the patterns at the input, one for each call.
    SetOut(j+count,0);
    if (mode == SIM)
        for (j=0; j<count; j++)
            SetOut(j,ipat[CurPat][j]);
    else
        for (j=0; j<count; j++)
            SetOut(j,tpat[CurPat][j]);
    CurPat++;
    CurPat %= count2;
}
```

```
void LearnAuto(){
Site *sp;
FLINT out, deltaW;
     for (int i=count; i<2*count; i++){
          out = GetOut(i);
          sp = GetSitePtr(i,"auto");
          for (Link *lp=GetLinks(sp); lp; lp=lp->next){
// find change in weight
               deltaW = ETA * out * GetLinkValue(lp);
               SetWeight(lp,GetWeight(lp)+deltaW);
          }
/*Data field of the unit is used for storing
                    the threshold for the unit.  */
// Adjust the threshold of the unit
               if (out==1)
                    SetUnitData(i,GetUnitData(i) + ETA);
               else
                    SetUnitData(i,GetUnitData(i) - ETA);
          }
}

UFauto(Unit *up){
Site *sp;
FLINT sum = 0;
     for (sp=GetSitesP(up); sp; sp=sp->next)
          sum += GetValue(sp);
     if (sum > GetDataP(up)){
          SetPotP(up,1);
          SetOutP(up,1);
     }
     else{
          SetPotP(up,-1);
          SetOutP(up,-1);
     }
}
```

```
PrintWts(){
     cout << "\n";
     for (int j=count; j<2*count; j++){
          Site *sp = GetSitePtr(j,"auto");
          for (Link *lp=GetLinks(sp); lp; lp=lp->next)
// Weights of all links from other auto associator units
               cout << form("%5d ",GetWeight(lp));
// Thresholds of auto associator units
          cout << form("   %5d",GetData(j));
          sp = GetSitePtr(j,"in");
          for (lp=GetLinks(sp); lp; lp=lp->next)
// Weights of links from other input units
               cout << form("   %5d",GetWeight(lp));
          cout << "\n";
     }
}
```

Note that, in this case, we have not written the network construction function (similar to the one in the pattern associator example). The following set of commands may be used for constructing, from scratch, t auto associator network. Note how the display control parameters have been used. For a small network with a simple topology as this, constructing the network from the command interface is alright. However, for larger or more complicated networks it is more easy to write a network construction functi as has been done in the case of other examples.

```
allocateunits 16
makeunit 8 INPUT Ø Ø Ø Ø Ø null Ø
addtoset Input Ø - 7
setunitfunction Input off
setsitefunction Input off
setlinkfunction Input off
makeunit 8 AUTO Ø Ø Ø Ø Ø UFauto Ø
addtoset Auto 8 - 15
setunitfunction Auto off
addsite Auto in SFweightedSum Ø
addsite Auto auto SFweightedSum Ø
makelink Auto Auto auto Ø Ø null
makelink Ø 8 in 1ØØØ Ø null
makelink 1 9 in 1ØØØ Ø null
makelink 2 1Ø in 1ØØØ Ø null
makelink 3 11 in 1ØØØ Ø null
makelink 4 12 in 1ØØØ Ø null
```

```
makelink 5 13 in 1000 0 null
makelink 6 14 in 1000 0 null
makelink 7 15 in 1000 0 null
setinputfunc ReadPat
setlearnfunc LearnAuto
list all on
list off
echo 5
pause
pipe
show
list
echo
sim 50 5
list on
test 6 5
call PrintWts
```

MAP COLORING PROBLEM

```
/* Functions for simulation of map coloring problem. */
#include <stdio.h>
#include "defs.h"
#include "extern.h"

#define STATIC 0
#define CHANGE 1
#define RED 0
#define BLUE 1
#define GREEN 2
#define WHITE 3
static char *colornames[] = {"red", "blue", "green", "white"};
int Count,b;

// returns sum of weight * value for all links at the site.
SFWeightedSum(Site *sp){
     int sum = 0;
     for(Link *lp=GetLinks(sp); lp; lp=lp->next)
          sum += GetLinkValue(lp) * GetWeight(lp);
     SetSiteValue(sp,sum / 1000);
}

// returns value 1 with probability (1000+inhibit)/1999;
// note that if inhibit <= 1000, this probability is zero.
inline int dice(int inhibit){
     return((random() % 1999) < (1000 + inhibit));
}

void UFcolor(Unit *up){
     int oldpot = GetPotP(up);    // save old potential
     int inhibit = SiteValue("inhibit",GetSitesP(up));
     if (inhibit >= 0 || dice(inhibit)){      // turn on unit
          SetPotP(up,1000);
          SetOutP(up,1000);
     }
     else{                                    // turn off unit
          SetPotP(up,0);
          SetOutP(up,0);
     }
 // change state if potential has changed since last time
     if (oldpot != GetPotP(up)){
          AddToSet("change",UnitIndex(up));
          SetStateP(up,CHANGE);
     }
     else
          if (MemberSet("change",UnitIndex(up))){
               RemFromSet("change",UnitIndex(up));
               SetStateP(up,STATIC);
          }
}
```

```
int region(){
static int regionnum = 0;
int i,j,first;
char buf[15];
int makecolor(int);
void mux(int, int, int);
     // make 4 consecutive units for the 4 colors
     first = makecolor(RED);
     makecolor(BLUE);
     makecolor(GREEN);
     makecolor(WHITE);
 // name the units by their region
     sprintf(buf,"region%1d",regionnum);
     NameUnit(buf,VECTOR,first,4);
     for (i=0; i<4; i++)            // set up inhibitory connections
          for (j=i+1; j<4; j++)
               mux(first+i,first+j,-1000);
     return regionnum++;
}

/* make an unit that represents a color for a region */
int makecolor(int type){
int index;
ufunc_ptr ufp;
sfunc_ptr sfp;
     ufp = UFcolor;
     sfp = SFWeightedSum;
     index = MakeUnit(colornames[type],STATIC,STATIC,0,0,0,ufp,0);
     MakeSite(index,"inhibit", sfp,0);
     SetFlag(index,NO_LINKFUNC_FLAG);
     return index;
}

/* set up bidirectional links */
void mux(int unit1, int unit2, int weight){
     MakeLink(unit1,unit2,"inhibit",weight,0,NULL);
     MakeLink(unit2,unit1,"inhibit",weight,0,NULL);
}

/* returns the index of unit for the given region and color */
int mapunit(int region, int color){
char buf[15];
     sprintf(buf,"region%1d",region);
     return NameToInd(buf,color);
}

/* make border between region1 and region2 */
void border(int region1, int region2){
void mux(int, int, int);
int mapunit(int, int);
     mux(mapunit(region1,BLUE),mapunit(region2,BLUE),-100);
      mux(mapunit(region1,RED),mapunit(region2,RED),-100);
     mux(mapunit(region1,GREEN),mapunit(region2,GREEN),-100);
     mux(mapunit(region1,WHITE),mapunit(region2,WHITE),-100);
}
```

```
void BuildNet(){
int count, i, reg1, reg2, readcount;
FILE *infile;
     if (argc != 3){          // Check number of arguments
          cout << "Usage: sim <data file>\n";
          return;
     }
     infile = fopen(argv[2],"r");   // open data file
     if (infile == NULL){
     cout << form("sim: could not open %s\n",argv[2]);
     return;
 }
     fscanf(infile,"%d",&count);  // get number of regions
     AllocateUnits(count*4);      // create space for regions * 4 units
// declare set for units which change state
     DeclareSet("change");
     DeclareState("Static",STATIC);   // declare states
     DeclareState("Change",CHANGE);

     for (i=0; i<count; i++)      // make regions
          region();
    Count = count;
     for (int j=0; ; j++){        // make borders
          readcount = fscanf(infile,"%d %d",&reg1,&reg2);
          if (readcount != 2)
               break;
          border(reg1,reg2);
     }
     b = j;
}

void PrintClrs(){
     cout << form("%d regions with %d borders",Count,b);
     for (int i=0; i<=CurIndex; i++)
     // print name & type (region & color) of active units
          if (GetPot(i))
               cout << form("%15s   %15s",GetName(i),GetType(i));
}

Commands for running the simulation:

call BuildNet <map_data>

list off

show on

show set + change

async

sim 50 2

call PrintClrs
```

THE DIPOLE EXPERIMENT

```
/* Functions for simulation of the Dipole Experiment. This
function the also illustrates the use of type FLINT which makes
it possible to use functions with either integer or
floating-point versions of the simulator */

#include "defs.h"
#include "extern.h"
#include "sim.h"

#ifdef FSIM
#define MAXWT 1.0
#else
#define MAXWT 1000
#endif

int ipat[24][2];
func_type SFweightedSum(Site *);
void UFdipole(Unit *);

void RandomWts(int i){
int k;
FLINT sum=0, tsum=0, temp[16];
     for (k=0; k<16; k++){
#ifdef FSIM
          temp[k] = rand01();
#else
          temp[k] = randint(0,1000);
#endif
          sum += temp[k];
     }
     for (k=0; k<16; k++){
```

```
            temp[k] /= sum;
            MakeLink(k,i,"feed",temp[k],Ø,NULL);
      }
}

void BuildNet(){
int i;

// Create space for input, output and target units.
      AllocateUnits(18);

// Make input units, indices Ø - 15.
      for (i=Ø; i<16; i++){
            MakeUnit("INPUT",Ø,Ø,Ø,Ø,Ø,NULL,Ø);
            SetFlag(i,NO_UNITFUNC_FLAG);     // No Unit function.
            SetFlag(i,NO_SITEFUNC_FLAG);     // No Site function.
            SetFlag(i,NO_LINKFUNC_FLAG);     // No Link function.
      }
      NameUnit("In",ARRAY,Ø,4,4);         // Name them as a vector - In.
      DeclareSet("InputSet");             // Make a set of the input units.
      AddToSet("InputSet",Ø,15);

// Make competing units, indices 16,17.
      for (i=16; i<18; i++){              // Make the competing units.
            MakeUnit("COMPETE",Ø,Ø,Ø,Ø,Ø,UFdipole,Ø);
            MakeSite(i,"feed",SFweightedSum,Ø);  // Add a site with name "feed"
            MakeSite(i,"inhibit",NULL,Ø);  // Add a site with name "inhibit".
            SetFlag(i,NO_LINKFUNC_FLAG);
}
      NameUnit("comp1",SCALAR,16);
      NameUnit("comp2",SCALAR,17);

// For each competing unit, make links from input units with random
//  weights such that sum of weights = 1.
      for (i=16; i<18; i++)
            RandomWts(i);
```

```
// Make inhibitory links between competing units, weight = -MAXWT.
     MakeLink(16,17,"inhibit",-MAXWT,0,NULL);
     MakeLink(17,16,"inhibit",-MAXWT,0,NULL);
}

void ReadPat(int mode){
static int readInput = FALSE;
FILE *infile;
int CurPat;
int i,j,n;
/* Though it is possible to generate the the dipole  patterns,  we
read them in for the sake of simplicity. */
     if (!readInput){         // To read the input file and create
                               // the arrays only
          readInput = TRUE;    // on the first call to this function.

          infile = fopen("dipole.dat","r");
          if (infile == NULL){
               cout << form("sim: could not open dipole.dat\n");
               return;
          }
          for (i=0; i<24; i++)      // Read in unit pairs that form dipoles.
               for (j=0; j<2; j++)
                    if (!(n = fscanf(infile,"%d ",&ipat[i][j]))){
                         cout << "Premature end of file.";
                         exit(1);
                    }
// Display the read patterns.
          cout << "Unit pairs that form dipoles:\n";
          for (i=0; i<24; i++){
               for (j=0; j<2; j++)
                    cout << form("%d ",ipat[i][j]);
               cout << "\n";
          }
          cout << "\n";
     }
```

```
// All of the above done only once.
// On each call, activate any one dipole at random,other outputs=0
     for (i=0; i<16; i++)
          SetOut(i,0);

     CurPat = randint(0,23);
     SetOut(ipat[CurPat][0],1);
     SetOut(ipat[CurPat][1],1);
}

float g = 0.1;          // Learning Rate
int n = 2;              // Number of active units at any time; since dipole

void LearnDipole(){
int winner;
FLINT deltaW;
Site *sp;
Link *lp;
     if (GetOut(16) > GetOut(17))  // indices of competing units = 16, 17
          winner = 16;
     else
          winner = 17;
     sp = GetSitePtr(winner,"feed");
     for (lp=GetLinks(sp); lp; lp=lp->next)
          if (GetLinkValue(lp)){
#ifdef FSIM
               deltaW = g * (1/(float)n - GetWeight(lp));
#else
               deltaW = g * (1000/(float)n - GetWeight(lp));
                         // scale up by a factor of 1000
#endif
               SetWeight(lp, GetWeight(lp)+deltaW);
          }
}
```

```
void UFdipole(Unit *up){
     Site *sp = GetSitePtrP(up,"feed");
     FLINT actvn = GetSiteValue(sp);
     SetPotP(up,actvn);
     SetOutP(up,actvn);
}

void PrintWts(){
int fromUnit;
     cout << "\n";
     cout << "Values for weights and outputs of COMP units:\n";
     for (int i=0; i<4; i++){
          for (int j=0; j<4; j++){
               fromUnit = 4 * i + j;
               for (int k=16; k<18; k++){
                    Site *sp = GetSitePtr(k,"feed");
                    Link *lp = GetLinkPtr(sp,fromUnit);
#ifdef FSIM
                    cout << form("%f %f  ",GetWeight(lp),GetOut(k));
#else
                    cout << form("%d %d  ",GetWeight(lp),GetOut(k));
#endif
               }
               cout << "\n";
          }
          cout << "";
     }
}
```

SYMMETRY DETECTOR

```
/* Functions for simulation of a Symmetry Detector Network.
   Uses Back-propagation Algorithm. This function does not use
   type FLINT for weights and outputs, instead uses 'float'.
   Hence, will work only with floating-point version of the
   simulator. */

#include "defs.h"
#include "extern.h"
#include "sim.h"
#define ETA 0.5
#define MAXWT 1.0
typedef int * intp;
intp *ipat;
int *tpat;
func_type SFweightedSum(Site *);
FILE *infile;
int inUnits,hidUnits,totUnits; // No. of input and hidden units.
                        // No. of output units = target units = 1
int patCount;                           // No. of patterns.

/* Function for constructing the network. */
void BuildNet(){
int i, j;
void UFback(Unit *up);
void SFweightedSum(Site *sp);
float rwt;

        if (argc !=3){
                printf("Usage: call BuildNet <file_name>\n");
                return;
        }

        infile = fopen(argv[2],"r");
        if (infile == NULL){
                cout << form("sim: could not open %s\n",argv[2]);
                return;
        }

// Get the number of units and the number of patterns.
        fscanf(infile,"%d %d %d\n",&inUnits,&hidUnits,&patCount);
        totUnits = inUnits + hidUnits;
        AllocateUnits(totUnits+2);

        for (i=0; i<inUnits; i++){
                MakeUnit("INPUT",0,0,0,0,0,NULL,0);
                SetFlag(i,NO_UNITFUNC_FLAG);
                SetFlag(i,NO_SITEFUNC_FLAG);
                SetFlag(i,NO_LINKFUNC_FLAG);
        }
```

```
        NameUnit("In",VECTOR,0,inUnits);
        DeclareSet("InputSet");
        AddToSet("InputSet",0,inUnits-1);

        for (i=inUnits; i<totUnits; i++){
                MakeUnit("HIDDEN",0,0,0,0,0,UFback,0);
                MakeSite(i,"feed",SFweightedSum,0);
                SetFlag(i,NO_LINKFUNC_FLAG);
        }
        NameUnit("Hid",VECTOR,inUnits,hidUnits);
        DeclareSet("HiddenSet");
        AddToSet("HiddenSet",inUnits,totUnits-1);

// index of output unit = totUnits
// index of target unit = totUnits+1

        i = MakeUnit("OUTPUT",0,0,0,0,0,UFback,0); // Make output unit
        MakeSite(i,"feed",SFweightedSum,0);
        MakeSite(i,"teach",NULL,0);
        SetFlag(i,NO_LINKFUNC_FLAG);
        NameUnit("Out",SCALAR,i);

        i = MakeUnit("TEACHER",0,0,0,0,0,NULL,0); // Make target unit.
        SetFlag(i,NO_UNITFUNC_FLAG);
        SetFlag(i,NO_SITEFUNC_FLAG);
        SetFlag(i,NO_LINKFUNC_FLAG);
        NameUnit("Teach",SCALAR,i);

// links from input units to "feed" at hidden units with
// random weights
        for (i=0; i<inUnits; i++)
                for (j=inUnits; j<totUnits; j++){
                        rwt = rand01();
                        if (randint(1,10) > 5)
                                rwt *= -1.0;
                        MakeLink(i,j,"feed",rwt,0,NULL);
                }
// links from hidden units to "feed" at output unit
// with random weights
        for (i=inUnits; i<totUnits; i++){
                rwt = rand01();
                if (randint(1,10) > 5)
                        rwt *= -1.0;
                MakeLink(i,totUnits,"feed",rwt,0,NULL);
        }
// Make link from target unit to site "teach" at output unit,
// weight = MAXWT.
        MakeLink(totUnits+1,totUnits,"teach",MAXWT,0,NULL);
```

```
// Set random biases on the hidden units.
        for (i=inUnits; i<=totUnits; i++){
                rwt = rand01();
                if (randint(1,10) > 5)
                        rwt *= -1.0;
                SetUnitData(i,rwt);
        }
}

void ReadPat(int mode){
static int readInput = FALSE;
static int CurPat=0;
int i,j,n;
// To read the input file and create the arrays only
// on the first call to this function.
        if (!readInput){
                readInput = TRUE;

                ipat = new intp[patCount];
                // Allocate array for input patterns...
                for (i=0; i<patCount; i++)
                        ipat[i] = new int[inUnits];

                // ...and the target patterns.
                tpat = new int[patCount];

// Read in the input patterns and the target patterns
// with check for sufficient data.
                // Read in the input patterns...
                for (i=0; i<patCount; i++){
                    for (j=0; j<inUnits; j++)
                        if (!(n = fscanf(infile,"%d ",&ipat[i][j]))){
                                cout << "Premature end of file.\n";
                                exit(1);
                        }
                        if (!(n = fscanf(infile,"%d ",&tpat[i]))){
                                cout << "Premature end of file.\n";
                                exit(1);
                        }
                }
// Display the read patterns.
                cout << "Input Pattern Set:\n";
                for (i=0; i<patCount; i++){
                        for (j=0; j<inUnits; j++)
                                cout << form("%d ",ipat[i][j]);
                        cout << form("  %d\n",tpat[i]);
                }
        }
// All of the above done only once.
```

```
// On each call set input vector on input units
// and 0 on output unit.
        for (j=0; j<inUnits; j++)
                SetOut(j,ipat[CurPat][j]);
        SetOut(totUnits,0);

// Set target vector on target units only if called
// during simulation.
        if (mode == SIM)
                SetOut(totUnits+1,tpat[CurPat]);
        else
                SetOut(totUnits+1,0);

        CurPat++;
        CurPat %= patCount;
        // To cycle over the same set of patterns.
}

/*      The Learning Rule.    */
void LearnBack(){
Site *sp;
Link *lp;
linkTo *ltp;
int i;
float dw,opj,dpj,tpj,dpk,wkj,sigmak;
        // find error for output units
        opj = GetOut(totUnits);
        tpj = GetOut(totUnits+1);
        dpj = (tpj - opj) * opj * (1 - opj);

        /* store error value in "feed"'s site-data field
        for lower layer units.  In this case, this superfluous;
        but can be helpful for networks with multiple hidden
        layers/multiple output units. */

        sp = GetSitePtr(totUnits,"feed");
        SetSiteData(sp,dpj);

        // update weights of links arriving at site "feed"
        // of output unit
        lp = GetLinks(sp);
        for ( ; lp; lp=lp->next){
                dw = ETA * dpj * GetLinkValue(lp);
                SetWeight(lp,GetWeight(lp) + dw);
        }
```

```
        /* In this case, all hidden units feed a common output unit
        whose index is known. Also, it is known that the connection from
        any hidden unit arrives at the site "feed" of the output unit.
        As such, there is no need to make use of a (hidden) unit's
        fan-out list to determine the error value of the unit to which
        it connects. For a more general case where there could be multiple
        output units or a network with multiple hidden layers, it becomes
        necessary to use the fan-out list to find the destination of
        each of the links originating from a unit in a lower layer. Thus,
        though we could have simply used the value of 'sp' determined above,
        we take the more general approach for the sake of illustration. */

        for (i=totUnits-1; i>=inUnits; i--){  // for each hidden unit
                sigmak = 0;
                for (ltp = GetFanOut(i); ltp; ltp=ltp->next){
                                        // for each unit on the fan-out list...
                        sp = GetSitePtr(ltp->toUnit,ltp->toSite);
                        dpk = GetSiteData(sp);  // ...get error value
                        wkj = GetWeight(GetLinkPtr(sp,i));
                        sigmak += dpk * wkj;
                }
                opj = GetOut(i);
                dpj = opj * (1 - opj) * sigmak;
                sp = GetSitePtr(i,"feed");
                SetSiteData(sp,dpj);  // set error at this unit, for use by lower
                                        // layer units(none, in this case).
                for (lp=GetLinks(sp); lp; lp=lp->next){
                                        // update weights of arriving links.
                        dw = ETA * dpj * GetLinkValue(lp);
                        SetWeight(lp,GetWeight(lp) + dw);
                }
        }
}

/* The Unit Activation function for the output units. */
void UFback(Unit *up){
Site *sp;
        sp = GetSitePtrP(up,"feed");
        float out = 1.0/(1.0 + exp(-1.0 *
                        (GetSiteValue(sp) + GetUnitDataP(up))));
                SetPotP(up,out);
                SetOutP(up,out);
}
```